

Perl Programmers Reference Guide

**Version 5.8.0
04-Aug-2002**

"There's more than one way to do it."

-- Larry Wall, Author of the Perl Programming Language

Author: Perl5-Porters

blank

NAME

perl – Practical Extraction and Report Language

SYNOPSIS

```
perl      [ -sTuU ] [ -hv ] [ -V[:configvar] ]
          [ -cw ] [ -d[:debugger] ] [ -D[number/list] ]
          [ -pna ] [ -Fpattern ] [ -I[octal] ] [ -O[octal] ]
          [ -Idir ] [ -m[-]module ] [ -M[-]'module...' ]
          [ -P ] [ -S ] [ -x[dir] ]
          [ -i[extension] ] [ -e 'command' ]
          [ — ] [ programfile ] [ argument ]...
```

If you're new to Perl, you should start with [perlintro](#), which is a general intro for beginners and provides some background to help you navigate the rest of Perl's extensive documentation.

For ease of access, the Perl manual has been split up into several sections.

Overview

perl	Perl overview (this section)
perlintro	Perl introduction for beginners
perltoc	Perl documentation table of contents

Tutorials

perlreftut	Perl references short introduction
perldsc	Perl data structures intro
perl101	Perl data structures: arrays of arrays
perlrequick	Perl regular expressions quick start
perlretut	Perl regular expressions tutorial
perlboot	Perl OO tutorial for beginners
perltoot	Perl OO tutorial, part 1
perltooc	Perl OO tutorial, part 2
perlbot	Perl OO tricks and examples
perlstyle	Perl style guide
perltrap	Perl traps for the unwary
perldebtut	Perl debugging tutorial
perlfaq	Perl frequently asked questions
perlfaq1	General Questions About Perl
perlfaq2	Obtaining and Learning about Perl
perlfaq3	Programming Tools
perlfaq4	Data Manipulation
perlfaq5	Files and Formats
perlfaq6	Regexes
perlfaq7	Perl Language Issues
perlfaq8	System Interaction
perlfaq9	Networking

Reference Manual

perlsyn	Perl syntax
perldata	Perl data structures
perlop	Perl operators and precedence
perlsub	Perl subroutines
perlfunc	Perl built-in functions
perlomentut	Perl open() tutorial
perlpacktut	Perl pack() and unpack() tutorial

perlpod	Perl plain old documentation
perlpodspec	Perl plain old documentation format specification
perlrun	Perl execution and options
perldiag	Perl diagnostic messages
perllexwarn	Perl warnings and their control
perldebug	Perl debugging
perlvar	Perl predefined variables
perlre	Perl regular expressions, the rest of the story
perlref	Perl references, the rest of the story
perlform	Perl formats
perlobj	Perl objects
perltie	Perl objects hidden behind simple variables
perldbmfilt	Perl DBM filters
perlipc	Perl interprocess communication
perlfork	Perl fork() information
perlnumber	Perl number semantics
perlthrtut	Perl threads tutorial
perlothrtut	Old Perl threads tutorial
perlport	Perl portability guide
perllocale	Perl locale support
perluniintro	Perl Unicode introduction
perlunicode	Perl Unicode support
perlebcdic	Considerations for running Perl on EBCDIC platforms
perlsec	Perl security
perlmod	Perl modules: how they work
perlmodlib	Perl modules: how to write and use
perlmodstyle	Perl modules: how to write modules with style
perlmodinstall	Perl modules: how to install from CPAN
perlnewmod	Perl modules: preparing a new module for distribution
perlutil	utilities packaged with the Perl distribution
perlcompile	Perl compiler suite intro
perlfiler	Perl source filters

Internals and C Language Interface

perlembed	Perl ways to embed perl in your C or C++ application
perldebbugs	Perl debugging guts and tips
perlxsut	Perl XS tutorial
perlxs	Perl XS application programming interface
perlclib	Internal replacements for standard C library functions
perlguts	Perl internal functions for those doing extensions
perlcall	Perl calling conventions from C
perlapi	Perl API listing (autogenerated)
perlintern	Perl internal functions (autogenerated)
perliol	C API for Perl's implementation of IO in Layers
perlapiio	Perl internal IO abstraction interface
perlhack	Perl hackers guide

Miscellaneous

perlbook	Perl book information
perltodo	Perl things to do


```

perlhist      Perl history records
perldelta      Perl changes since previous version
perl572deltaPerl changes in version 5.7.2
perl571deltaPerl changes in version 5.7.1
perl570deltaPerl changes in version 5.7.0
perl561deltaPerl changes in version 5.6.1
perl56delta      Perl changes in version 5.6
perl5005deltaPerl changes in version 5.005
perl5004deltaPerl changes in version 5.004

```

Language-Specific

```

perlcn          Perl for Simplified Chinese (in EUC-CN)
perljp          Perl for Japanese (in EUC-JP)
perlko          Perl for Korean (in EUC-KR)
perltw          Perl for Traditional Chinese (in Big5)

```

Platform-Specific

```

perlaix          Perl notes for AIX
perlamiga        Perl notes for AmigaOS
perlapollo       Perl notes for Apollo DomainOS
perlbeos         Perl notes for BeOS
perlbs2000       Perl notes for POSIX-BC BS2000
perlce           Perl notes for WinCE
perlcygwin       Perl notes for Cygwin
perldgux         Perl notes for DG/UX
perldos          Perl notes for DOS
perlepoc         Perl notes for EPOC
perlfreebsd      Perl notes for FreeBSD
perlhpx          Perl notes for HP-UX
perlhurd         Perl notes for Hurd
perlirix         Perl notes for Irix
perlmachten      Perl notes for Power MachTen
perlmacos        Perl notes for Mac OS (Classic)
perlmint         Perl notes for MiNT
perlmpaix        Perl notes for MPE/iX
perlnetware      Perl notes for NetWare
perlos2          Perl notes for OS/2
perlos390        Perl notes for OS/390
perlplan9        Perl notes for Plan 9
perlqnx          Perl notes for QNX
perlsolaris      Perl notes for Solaris
perltru64        Perl notes for Tru64
perluts          Perl notes for UTS
perlvmsa         Perl notes for VM/ESA
perlvms          Perl notes for VMS
perlvos          Perl notes for Stratus VOS
perlwin32        Perl notes for Windows

```

By default, the manpages listed above are installed in the `/usr/local/man/` directory.

Extensive additional documentation for Perl modules is available. The default configuration for perl will place this additional documentation in the `/usr/local/lib/perl5/man` directory (or else in the `man` subdirectory of the Perl library directory). Some of this additional documentation is distributed standard with Perl, but you'll also find documentation for third-party modules there.

You should be able to view Perl's documentation with your `man(1)` program by including the proper directories in the appropriate start-up files, or in the `MANPATH` environment variable. To find out where

the configuration has installed the manpages, type:

```
perl -V:man.dir
```

If the directories have a common stem, such as `/usr/local/man/man1` and `/usr/local/man/man3`, you need only to add that stem (`/usr/local/man`) to your `man(1)` configuration files or your `MANPATH` environment variable. If they do not share a stem, you'll have to add both stems.

If that doesn't work for some reason, you can still use the supplied *perldoc* script to view module information. You might also look into getting a replacement man program.

If something strange has gone wrong with your program and you're not sure where you should look for help, try the `-w` switch first. It will often point out exactly where the trouble is.

DESCRIPTION

Perl is a language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal).

Perl combines (in the author's opinion, anyway) some of the best features of C, **sed**, **awk**, and **sh**, so people familiar with those languages should have little difficulty with it. (Language historians will also note some vestiges of **csh**, Pascal, and even BASIC-PLUS.) Expression syntax corresponds closely to C expression syntax. Unlike most Unix utilities, Perl does not arbitrarily limit the size of your data—if you've got the memory, Perl can slurp in your whole file as a single string. Recursion is of unlimited depth. And the tables used by hashes (sometimes called "associative arrays") grow as necessary to prevent degraded performance. Perl can use sophisticated pattern matching techniques to scan large amounts of data quickly. Although optimized for scanning text, Perl can also deal with binary data, and can make dbm files look like hashes. Setuid Perl scripts are safer than C programs through a dataflow tracing mechanism that prevents many stupid security holes.

If you have a problem that would ordinarily use **sed** or **awk** or **sh**, but it exceeds their capabilities or must run a little faster, and you don't want to write the silly thing in C, then Perl may be for you. There are also translators to turn your **sed** and **awk** scripts into Perl scripts.

But wait, there's more...

Begun in 1993 (see *perlhist*), Perl version 5 is nearly a complete rewrite that provides the following additional benefits:

- modularity and reusability using innumerable modules
Described in *perlmod*, *perlmodlib*, and *perlmodinstall*.
- embeddable and extensible
Described in *perlembed*, *perlxtut*, *perlxs*, *perlcalls*, *perlguts*, and *xsubpp*.
- roll-your-own magic variables (including multiple simultaneous DBM implementations)
Described in *perltie* and *AnyDBM_File*.
- subroutines can now be overridden, autoloaded, and prototyped
Described in *perlsub*.
- arbitrarily nested data structures and anonymous functions
Described in *perlreftut*, *perlref*, *perldsc*, and *perllool*.
- object-oriented programming
Described in *perlobj*, *perlboot*, *perltoot*, *perltooc*, and *perlbot*.

- compilability into C code or Perl bytecode
Described in [B](#) and [B::Bytecode](#).
- support for light-weight processes (threads)
Described in [perlthrtut](#) and [Thread](#).
- support for internationalization, localization, and Unicode
Described in [perllocale](#) and [utf8](#).
- lexical scoping
Described in [perlsub](#).
- regular expression enhancements
Described in [perlre](#), with additional examples in [perlop](#).
- enhanced debugger and interactive Perl environment, with integrated editor support
Described in [perldebtut](#), [perldebug](#) and [perldebguts](#).
- POSIX 1003.1 compliant library
Described in [POSIX](#).

Okay, that's *definitely* enough hype.

AVAILABILITY

Perl is available for most operating systems, including virtually all Unix-like platforms. See [Supported Platforms in perlport](#) for a listing.

ENVIRONMENT

See [perlrun](#).

AUTHOR

Larry Wall <larry@wall.org, with the help of oodles of other folks.

If your Perl success stories and testimonials may be of help to others who wish to advocate the use of Perl in their applications, or if you wish to simply express your gratitude to Larry and the Perl developers, please write to perl-thanks@perl.org .

FILES

"@INC" locations of perl libraries

SEE ALSO

a2p awk to perl translator
s2p sed to perl translator

<http://www.perl.com/> the Perl Home Page
<http://www.cpan.org/> the Comprehensive Perl Archive
<http://www.perl.org/> Perl Mongers (Perl user groups)

DIAGNOSTICS

The use warnings pragma (and the `-w` switch) produces some lovely diagnostics.

See [perldiag](#) for explanations of all Perl's diagnostics. The use diagnostics pragma automatically turns Perl's normally terse warnings and errors into these longer forms.

Compilation errors will tell you the line number of the error, with an indication of the next token or token type that was to be examined. (In a script passed to Perl via `-e` switches, each `-e` is counted as one line.)

Setuid scripts have additional constraints that can produce error messages such as "Insecure dependency".

See [perlsec](#).

Did we mention that you should definitely consider using the `-w` switch?

BUGS

The `-w` switch is not mandatory.

Perl is at the mercy of your machine's definitions of various operations such as type casting, `atof()`, and floating-point output with `sprintf()`.

If your `stdio` requires a seek or eof between reads and writes on a particular stream, so does Perl. (This doesn't apply to `sysread()` and `syswrite()`.)

While none of the built-in data types have any arbitrary size limits (apart from memory size), there are still a few arbitrary limits: a given variable name may not be longer than 251 characters. Line numbers displayed by diagnostics are internally stored as short integers, so they are limited to a maximum of 65535 (higher numbers usually being affected by wraparound).

You may mail your bug reports (be sure to include full configuration information as output by the `myconfig` program in the perl source tree, or by `perl -V`) to `perlbug@perl.org`. If you've succeeded in compiling perl, the **perlbug** script in the *utils/* subdirectory can be used to help mail in a bug report.

Perl actually stands for Pathologically Eclectic Rubbish Lister, but don't tell anyone I said that.

NOTES

The Perl motto is "There's more than one way to do it." Divining how many more is left as an exercise to the reader.

The three principal virtues of a programmer are Laziness, Impatience, and Hubris. See the Camel Book for why.

NAME

perl5004delta – what's new for perl5.004

DESCRIPTION

This document describes differences between the 5.003 release (as documented in *Programming Perl*, second edition—the Camel Book) and this one.

Supported Environments

Perl5.004 builds out of the box on Unix, Plan 9, LynxOS, VMS, OS/2, QNX, AmigaOS, and Windows NT. Perl runs on Windows 95 as well, but it cannot be built there, for lack of a reasonable command interpreter.

Core Changes

Most importantly, many bugs were fixed, including several security problems. See the *Changes* file in the distribution for details.

List assignment to %ENV works

%ENV = () and %ENV = @list now work as expected (except on VMS where it generates a fatal error).

Change to "Can't locate Foo.pm in @INC" error

The error "Can't locate Foo.pm in @INC" now lists the contents of @INC for easier debugging.

Compilation option: Binary compatibility with 5.003

There is a new Configure question that asks if you want to maintain binary compatibility with Perl 5.003. If you choose binary compatibility, you do not have to recompile your extensions, but you might have symbol conflicts if you embed Perl in another application, just as in the 5.003 release. By default, binary compatibility is preserved at the expense of symbol table pollution.

\$PERL5OPT environment variable

You may now put Perl options in the \$PERL5OPT environment variable. Unless Perl is running with taint checks, it will interpret this variable as if its contents had appeared on a "#!perl" line at the beginning of your script, except that hyphens are optional. PERL5OPT may only be used to set the following switches: `-[DIMUdmw]`.

Limitations on -M, -m, and -T options

The `-M` and `-m` options are no longer allowed on the `#!` line of a script. If a script needs a module, it should invoke it with the `use` pragma.

The `-T` option is also forbidden on the `#!` line of a script, unless it was present on the Perl command line. Due to the way `#!` works, this usually means that `-T` must be in the first argument. Thus:

```
#!/usr/bin/perl -T -w
```

will probably work for an executable script invoked as `scriptname`, while:

```
#!/usr/bin/perl -w -T
```

will probably fail under the same conditions. (Non-Unix systems will probably not follow this rule.) But `perl scriptname` is guaranteed to fail, since then there is no chance of `-T` being found on the command line before it is found on the `#!` line.

More precise warnings

If you removed the `-w` option from your Perl 5.003 scripts because it made Perl too verbose, we recommend that you try putting it back when you upgrade to Perl 5.004. Each new perl version tends to remove some undesirable warnings, while adding new warnings that may catch bugs in your scripts.

Deprecated: Inherited AUTOLOAD for non-methods

Before Perl 5.004, AUTOLOAD functions were looked up as methods (using the @ISA hierarchy), even when the function to be autoloaded was called as a plain function (e.g. `Foo::bar()`), not a method (e.g. `<`

`Foo-bar()` or `< $obj-bar()`).

Perl 5.005 will use method lookup only for methods' AUTOLOADs. However, there is a significant base of existing code that may be using the old behavior. So, as an interim step, Perl 5.004 issues an optional warning when a non-method uses an inherited AUTOLOAD.

The simple rule is: Inheritance will not work when autoloading non-methods. The simple fix for old code is: In any module that used to depend on inheriting AUTOLOAD for non-methods from a base class named `BaseClass`, execute `*AUTOLOAD = \&BaseClass::AUTOLOAD` during startup.

Previously deprecated %OVERLOAD is no longer usable

Using %OVERLOAD to define overloading was deprecated in 5.003. Overloading is now defined using the overload pragma. %OVERLOAD is still used internally but should not be used by Perl scripts. See [overload](#) for more details.

Subroutine arguments created only when they're modified

In Perl 5.004, nonexistent array and hash elements used as subroutine parameters are brought into existence only if they are actually assigned to (via `@_.`).

Earlier versions of Perl vary in their handling of such arguments. Perl versions 5.002 and 5.003 always brought them into existence. Perl versions 5.000 and 5.001 brought them into existence only if they were not the first argument (which was almost certainly a bug). Earlier versions of Perl never brought them into existence.

For example, given this code:

```
undef @a; undef %a;
sub show { print $_[0] };
sub change { $_[0]++ };
show($a[2]);
change($a{b});
```

After this code executes in Perl 5.004, `$a{b}` exists but `$a[2]` does not. In Perl 5.002 and 5.003, both `$a{b}` and `$a[2]` would have existed (but `$a[2]`'s value would have been undefined).

Group vector changeable with \$)

The `$)` special variable has always (well, in Perl 5, at least) reflected not only the current effective group, but also the group list as returned by the `getgroups()` C function (if there is one). However, until this release, there has not been a way to call the `setgroups()` C function from Perl.

In Perl 5.004, assigning to `$)` is exactly symmetrical with examining it: The first number in its string value is used as the effective gid; if there are any numbers after the first one, they are passed to the `setgroups()` C function (if there is one).

Fixed parsing of \$\$<digit, &\$<digit, etc.

Perl versions before 5.004 misinterpreted any type marker followed by "\$" and a digit. For example, "\$\$0" was incorrectly taken to mean "\${\$}0" instead of "\${\$0}" . This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "\$\$0" in a string. So Perl 5.004 still interprets "\$\$<digit" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

Fixed localization of \$<digit, \$&, etc.

Perl versions before 5.004 did not always properly localize the regex-related special variables. Perl 5.004 does localize them, as the documentation has always said it should. This may result in `$1`, `$2`, etc. no longer being set where existing programs use them.

No resetting of `$.` on implicit close

The documentation for Perl 5.0 has always stated that `$.` is *not* reset when an already-open file handle is reopened with no intervening call to `close`. Due to a bug, perl versions 5.000 through 5.003 *did* reset `$.` under that circumstance; Perl 5.004 does not.

`wantarray` may return undef

The `wantarray` operator returns true if a subroutine is expected to return a list, and false otherwise. In Perl 5.004, `wantarray` can also return the undefined value if a subroutine's return value will not be used at all, which allows subroutines to avoid a time-consuming calculation of a return value if it isn't going to be used.

`eval` **EXPR** determines value of **EXPR** in scalar context

Perl (version 5) used to determine the value of **EXPR** inconsistently, sometimes incorrectly using the surrounding context for the determination. Now, the value of **EXPR** (before being parsed by `eval`) is always determined in a scalar context. Once parsed, it is executed as before, by providing the context that the scope surrounding the `eval` provided. This change makes the behavior Perl4 compatible, besides fixing bugs resulting from the inconsistent behavior. This program:

```
@a = qw(time now is time);
print eval @a;
print '|', scalar eval @a;
```

used to print something like "timenowis881399109|4", but now (and in perl4) prints "4|4".

Changes to tainting checks

A bug in previous versions may have failed to detect some insecure conditions when taint checks are turned on. (Taint checks are used in `setuid` or `setgid` scripts, or when explicitly turned on with the `-T` invocation option.) Although it's unlikely, this may cause a previously-working script to now fail — which should be construed as a blessing, since that indicates a potentially-serious security hole was just plugged.

The new restrictions when tainting include:

No `glob()` or `<*`

These operators may spawn the C shell (`csh`), which cannot be made safe. This restriction will be lifted in a future version of Perl when globbing is implemented without the use of an external program.

No spawning if tainted `$CDPATH`, `$ENV`, `$BASH_ENV`

These environment variables may alter the behavior of spawned programs (especially shells) in ways that subvert security. So now they are treated as dangerous, in the manner of `$IFS` and `$PATH`.

No spawning if tainted `$TERM` doesn't look like a terminal name

Some termcap libraries do unsafe things with `$TERM`. However, it would be unnecessarily harsh to treat all `$TERM` values as unsafe, since only shell metacharacters can cause trouble in `$TERM`. So a tainted `$TERM` is considered to be safe if it contains only alphanumerics, underscores, dashes, and colons, and unsafe if it contains other characters (including whitespace).

New Opcode module and revised Safe module

A new Opcode module supports the creation, manipulation and application of opcode masks. The revised Safe module has a new API and is implemented using the new Opcode module. Please read the new Opcode and Safe documentation.

Embedding improvements

In older versions of Perl it was not possible to create more than one Perl interpreter instance inside a single process without leaking like a sieve and/or crashing. The bugs that caused this behavior have all been fixed. However, you still must take care when embedding Perl in a C program. See the updated `perlembed` manpage for tips on how to manage your interpreters.

Internal change: FileHandle class based on IO::* classes

File handles are now stored internally as type `IO::Handle`. The `FileHandle` module is still supported for backwards compatibility, but it is now merely a front end to the `IO::*` modules — specifically, `IO::Handle`, `IO::Seekable`, and `IO::File`. We suggest, but do not require, that you use the `IO::*` modules in new code.

In harmony with this change, `*GLOB{FILEHANDLE}` is now just a backward-compatible synonym for `*GLOB{IO}`.

Internal change: PerlIO abstraction interface

It is now possible to build Perl with AT&T's `sfio` IO package instead of `stdio`. See [perlapi](#) for more details, and the *INSTALL* file for how to use it.

New and changed syntax**`$coderef-(PARAMS)`**

A subroutine reference may now be suffixed with an arrow and a (possibly empty) parameter list. This syntax denotes a call of the referenced subroutine, with the given parameters (if any).

This new syntax follows the pattern of `< $hashref-{FOO}` and `< $aryref-[$foo]` : You may now write `&$subref($foo)` as `< $subref-($foo)` . All these arrow terms may be chained; thus, `< &{$stable-{FOO}}($bar)` may now be written `< $stable-{FOO}-($bar)` .

New and changed builtin constants**`__PACKAGE__`**

The current package name at compile time, or the undefined value if there is no current package (due to a `package;` directive). Like `__FILE__` and `__LINE__`, `__PACKAGE__` does *not* interpolate into strings.

New and changed builtin variables

`$_E` Extended error message on some platforms. (Also known as `$EXTENDED_OS_ERROR` if you use English).

`$_H` The current set of syntax checks enabled by `use strict`. See the documentation of `strict` for more details. Not actually new, but newly documented. Because it is intended for internal use by Perl core components, there is no `use English` long name for this variable.

`$_M` By default, running out of memory it is not trappable. However, if compiled for this, Perl may use the contents of `$_M` as an emergency pool after `die()`ing with this message. Suppose that your Perl were compiled with `-DPERL_EMERGENCY_SBRK` and used Perl's `malloc`. Then

```
$_M = 'a' x (1<<16);
```

would allocate a 64K buffer for use when in emergency. See the *INSTALL* file for information on how to enable this option. As a disincentive to casual use of this advanced feature, there is no `use English` long name for this variable.

New and changed builtin functions**`delete` on slices**

This now works. (e.g. `delete @ENV{'PATH', 'MANPATH'}`)

`flock`

is now supported on more platforms, prefers `fcntl` to `lockf` when emulating, and always flushes before (un)locking.

`printf` and `sprintf`

Perl now implements these functions itself; it doesn't use the C library function `sprintf()` any more, except for floating-point numbers, and even then only known flags are allowed. As a result, it is now possible to know which conversions and flags will work, and what they will do.

The new conversions in Perl's `sprintf()` are:

```
%i    a synonym for %d
%p    a pointer (the address of the Perl value, in hexadecimal)
%n    special: *stores* the number of characters output so far
      into the next variable in the parameter list
```

The new flags that go between the % and the conversion are:

```
#    prefix octal with "0", hex with "0x"
h    interpret integer as C type "short" or "unsigned short"
V    interpret integer as Perl's standard integer type
```

Also, where a number would appear in the flags, an asterisk ("`*`") may be used instead, in which case Perl uses the next item in the parameter list as the given number (that is, as the field width or precision). If a field width obtained through "`*`" is negative, it has the same effect as the "`-`" flag: left-justification.

See *sprintf* for a complete list of conversion and flags.

keys as an lvalue

As an lvalue, `keys` allows you to increase the number of hash buckets allocated for the given hash. This can gain you a measure of efficiency if you know the hash is going to get big. (This is similar to pre-extending an array by assigning a larger number to `$#array`.) If you say

```
keys %hash = 200;
```

then `%hash` will have at least 200 buckets allocated for it. These buckets will be retained even if you do `%hash = ()`; use `undef %hash` if you want to free the storage while `%hash` is still in scope. You can't shrink the number of buckets allocated for the hash using `keys` in this way (but you needn't worry about doing this by accident, as trying has no effect).

my() in Control Structures

You can now use `my()` (with or without the parentheses) in the control expressions of control structures such as:

```
while (defined(my $line = <>)) {
    $line = lc $line;
} continue {
    print $line;
}

if ((my $answer = <STDIN>) =~ /^y(es)?$/i) {
    user_agrees();
} elsif ($answer =~ /^n(o)?$/i) {
    user_disagrees();
} else {
    chomp $answer;
    die "`$answer' is neither 'yes' nor 'no'";
}
```

Also, you can declare a `foreach` loop control variable as lexical by preceding it with the word "`my`". For example, in:

```
foreach my $i (1, 2, 3) {
    some_function();
}
```

`$i` is a lexical variable, and the scope of `$i` extends to the end of the loop, but not beyond it.

Note that you still cannot use `my()` on global punctuation variables such as `$_` and the like.

pack() and unpack()

A new format 'w' represents a BER compressed integer (as defined in ASN.1). Its format is a sequence of one or more bytes, each of which provides seven bits of the total value, with the most significant first. Bit eight of each byte is set, except for the last byte, in which bit eight is clear.

If 'p' or 'P' are given undef as values, they now generate a NULL pointer.

Both pack() and unpack() now fail when their templates contain invalid types. (Invalid types used to be ignored.)

sysseek()

The new sysseek() operator is a variant of seek() that sets and gets the file's system read/write position, using the lseek(2) system call. It is the only reliable way to seek before using sysread() or syswrite(). Its return value is the new position, or the undefined value on failure.

use VERSION

If the first argument to use is a number, it is treated as a version number instead of a module name. If the version of the Perl interpreter is less than VERSION, then an error message is printed and Perl exits immediately. Because use occurs at compile time, this check happens immediately during the compilation process, unlike require VERSION, which waits until runtime for the check. This is often useful if you need to check the current Perl version before using library modules which have changed in incompatible ways from older versions of Perl. (We try not to do this more than we have to.)

use Module VERSION LIST

If the VERSION argument is present between Module and LIST, then the use will call the VERSION method in class Module with the given version as an argument. The default VERSION method, inherited from the UNIVERSAL class, croaks if the given version is larger than the value of the variable \$Module::VERSION. (Note that there is not a comma after VERSION!)

This version-checking mechanism is similar to the one currently used in the Exporter module, but it is faster and can be used with modules that don't use the Exporter. It is the recommended method for new code.

prototype(FUNCTION)

Returns the prototype of a function as a string (or undef if the function has no prototype). FUNCTION is a reference to or the name of the function whose prototype you want to retrieve. (Not actually new; just never documented before.)

srand

The default seed for srand, which used to be time, has been changed. Now it's a heady mix of difficult-to-predict system-dependent values, which should be sufficient for most everyday purposes.

Previous to version 5.004, calling rand without first calling srand would yield the same sequence of random numbers on most or all machines. Now, when perl sees that you're calling rand and haven't yet called srand, it calls srand with the default seed. You should still call srand manually if your code might ever be run on a pre-5.004 system, of course, or if you want a seed other than the default.

\$_ as Default

Functions documented in the Camel to default to \$_ now in fact do, and all those that do are so documented in *perlfunc*.

m//gc does not reset search position on failure

The m//g match iteration construct has always reset its target string's search position (which is visible through the pos operator) when a match fails; as a result, the next m//g match after a failure starts again at the beginning of the string. With Perl 5.004, this reset may be disabled by adding the "c" (for "continue") modifier, i.e. m//gc. This feature, in conjunction with the \G zero-width assertion, makes it possible to chain matches together. See *perlop* and *perlre*.

m//x ignores whitespace before ?*+{ }

The `m//x` construct has always been intended to ignore all unescaped whitespace. However, before Perl 5.004, whitespace had the effect of escaping repeat modifiers like `"*"` or `"?"`; for example, `/a*b/x` was (mis)interpreted as `/a*b/x`. This bug has been fixed in 5.004.

nested sub{ } closures work now

Prior to the 5.004 release, nested anonymous functions didn't work right. They do now.

formats work right on changing lexicals

Just like anonymous functions that contain lexical variables that change (like a lexical index variable for a `foreach` loop), formats now work properly. For example, this silently failed before (printed only zeros), but is fine now:

```
my $i;
foreach $i ( 1 .. 10 ) {
    write;
}
format =
    my i is @#
    $i
.
```

However, it still fails (without a warning) if the `foreach` is within a subroutine:

```
my $i;
sub foo {
    foreach $i ( 1 .. 10 ) {
        write;
    }
}
foo;
format =
    my i is @#
    $i
.
```

New builtin methods

The `UNIVERSAL` package automatically contains the following methods that are inherited by all other classes:

isa(CLASS)

`isa` returns *true* if its object is blessed into a subclass of `CLASS`

`isa` is also exportable and can be called as a sub with two arguments. This allows the ability to check what a reference points to. Example:

```
use UNIVERSAL qw(isa);

if(isa($ref, 'ARRAY')) {
    ...
}
```

can(METHOD)

`can` checks to see if its object has a method called `METHOD`, if it does then a reference to the sub is returned; if it does not then *undef* is returned.

VERSION([NEED])

`VERSION` returns the version number of the class (package). If the `NEED` argument is given then it will check that the current version (as defined by the `$VERSION` variable in the given package) not

less than `NEED`; it will die if this is not the case. This method is normally called as a class method. This method is called automatically by the `VERSION` form of `use`.

```
use A 1.2 qw(some imported subs);
# implies:
A->VERSION(1.2);
```

NOTE: `can` directly uses Perl's internal code for method lookup, and `isa` uses a very similar method and caching strategy. This may cause strange effects if the Perl code dynamically changes `@ISA` in any package.

You may add other methods to the `UNIVERSAL` class via Perl or XS code. You do not need to use `UNIVERSAL` in order to make these methods available to your program. This is necessary only if you wish to have `isa` available as a plain subroutine in the current package.

TIEHANDLE now supported

See [perltie](#) for other kinds of `tie()`s.

TIEHANDLE classname, LIST

This is the constructor for the class. That means it is expected to return an object of some sort. The reference can be used to hold some internal information.

```
sub TIEHANDLE {
    print "<shout>\n";
    my $i;
    return bless \$i, shift;
}
```

PRINT this, LIST

This method will be triggered every time the tied handle is printed to. Beyond its self reference it also expects the list that was passed to the print function.

```
sub PRINT {
    $r = shift;
    $$r++;
    return print join( $, => map {uc} @_), $\;
}
```

PRINTF this, LIST

This method will be triggered every time the tied handle is printed to with the `printf()` function. Beyond its self reference it also expects the format and list that was passed to the `printf` function.

```
sub PRINTF {
    shift;
    my $fmt = shift;
    print sprintf($fmt, @_)."\n";
}
```

READ this LIST

This method will be called when the handle is read from via the `read` or `sysread` functions.

```
sub READ {
    $r = shift;
    my ($buf,$len,$offset) = @_;
    print "READ called, \$buf=$buf, \$len=$len, \$offset=$offset";
}
```

READLINE this

This method will be called when the handle is read from. The method should return `undef` when there is no more data.

```
sub READLINE {
    $r = shift;
    return "PRINT called $$r times\n"
}
```

GETC this

This method will be called when the `getc` function is called.

```
sub GETC { print "Don't GETC, Get Perl"; return "a"; }
```

DESTROY this

As with the other types of ties, this method will be called when the tied handle is about to be destroyed. This is useful for debugging and possibly for cleaning up.

```
sub DESTROY {
    print "</shout>\n";
}
```

Malloc enhancements

If perl is compiled with the malloc included with the perl distribution (that is, if `perl -V:d_mymalloc` is 'define') then you can print memory statistics at runtime by running Perl thusly:

```
env PERL_DEBUG_MSTATS=2 perl your_script_here
```

The value of 2 means to print statistics after compilation and on exit; with a value of 1, the statistics are printed only on exit. (If you want the statistics at an arbitrary time, you'll need to install the optional module `Devel::Peek`.)

Three new compilation flags are recognized by `malloc.c`. (They have no effect if perl is compiled with `system malloc()`.)

-DPERL_EMERGENCY_SBRK

If this macro is defined, running out of memory need not be a fatal error: a memory pool can be allocated by assigning to the special variable `$^M`. See "[\\$^M](#)".

-DPACK_MALLOC

Perl memory allocation is by bucket with sizes close to powers of two. Because of these malloc overhead may be big, especially for data of size exactly a power of two. If `PACK_MALLOC` is defined, perl uses a slightly different algorithm for small allocations (up to 64 bytes long), which makes it possible to have overhead down to 1 byte for allocations which are powers of two (and appear quite often).

Expected memory savings (with 8-byte alignment in `alignbytes`) is about 20% for typical Perl usage. Expected slowdown due to additional malloc overhead is in fractions of a percent (hard to measure, because of the effect of saved memory on speed).

-DTWO_POT_OPTIMIZE

Similarly to `PACK_MALLOC`, this macro improves allocations of data with size close to a power of two; but this works for big allocations (starting with 16K by default). Such allocations are typical for big hashes and special-purpose scripts, especially image processing.

On recent systems, the fact that perl requires 2M from system for 1M allocation will not affect speed of execution, since the tail of such a chunk is not going to be touched (and thus will not require real memory). However, it may result in a premature out-of-memory error. So if you will be manipulating very large blocks with sizes close to powers of two, it would be wise to define this macro.

Expected saving of memory is 0–100% (100% in applications which require most memory in such `2**n` chunks); expected slowdown is negligible.

Miscellaneous efficiency enhancements

Functions that have an empty prototype and that do nothing but return a fixed value are now inlined (e.g. `sub PI () { 3.14159 }`).

Each unique hash key is only allocated once, no matter how many hashes have an entry with that key. So even if you have 100 copies of the same hash, the hash keys never have to be reallocated.

Support for More Operating Systems

Support for the following operating systems is new in Perl 5.004.

Win32

Perl 5.004 now includes support for building a "native" perl under Windows NT, using the Microsoft Visual C++ compiler (versions 2.0 and above) or the Borland C++ compiler (versions 5.02 and above). The resulting perl can be used under Windows 95 (if it is installed in the same directory locations as it got installed in Windows NT). This port includes support for perl extension building tools like *MakeMaker* and *h2xs*, so that many extensions available on the Comprehensive Perl Archive Network (CPAN) can now be readily built under Windows NT. See <http://www.perl.com/> for more information on CPAN and *README.win32* in the perl distribution for more details on how to get started with building this port.

There is also support for building perl under the Cygwin32 environment. Cygwin32 is a set of GNU tools that make it possible to compile and run many Unix programs under Windows NT by providing a mostly Unix-like interface for compilation and execution. See *README.cygwin32* in the perl distribution for more details on this port and how to obtain the Cygwin32 toolkit.

Plan 9

See *README.plan9* in the perl distribution.

QNX

See *README.qnx* in the perl distribution.

AmigaOS

See *README.amigaos* in the perl distribution.

Pragmata

Six new pragmatic modules exist:

`use autouse MODULE = qw(sub1 sub2 sub3)`

Defers `require MODULE` until someone calls one of the specified subroutines (which must be exported by `MODULE`). This pragma should be used with caution, and only when necessary.

`use blib`

`use blib 'dir'`

Looks for MakeMaker-like *'blib'* directory structure starting in *dir* (or current directory) and working back up to five levels of parent directories.

Intended for use on command line with `-M` option as a way of testing arbitrary scripts against an uninstalled version of a package.

`use constant NAME = VALUE`

Provides a convenient interface for creating compile-time constants, See *Constant Functions in perlsub*.

`use locale`

Tells the compiler to enable (or disable) the use of POSIX locales for builtin operations.

When `use locale` is in effect, the current `LC_CTYPE` locale is used for regular expressions and case mapping; `LC_COLLATE` for string ordering; and `LC_NUMERIC` for numeric formatting in `printf` and `sprintf` (but **not** in `print`). `LC_NUMERIC` is always used in `write`, since lexical scoping of formats is problematic at best.

Each use `locale` or no `locale` affects statements to the end of the enclosing BLOCK or, if not inside a BLOCK, to the end of the current file. Locales can be switched and queried with `POSIX::setlocale()`.

See [perllocale](#) for more information.

use ops

Disable unsafe opcodes, or any named opcodes, when compiling Perl code.

use vmsish

Enable VMS-specific language features. Currently, there are three VMS-specific features available: 'status', which makes `$?` and `system` return genuine VMS status values instead of emulating POSIX; 'exit', which makes `exit` take a genuine VMS status value instead of assuming that `exit 1` is an error; and 'time', which makes all times relative to the local time zone, in the VMS tradition.

Modules

Required Updates

Though Perl 5.004 is compatible with almost all modules that work with Perl 5.003, there are a few exceptions:

Module	Required Version for Perl 5.004
-----	-----
Filter	Filter-1.12
LWP	libwww-perl-5.08
Tk	Tk400.202 (-w makes noise)

Also, the majordomo mailing list program, version 1.94.1, doesn't work with Perl 5.004 (nor with perl 4), because it executes an invalid regular expression. This bug is fixed in majordomo version 1.94.2.

Installation directories

The *installperl* script now places the Perl source files for extensions in the architecture-specific library directory, which is where the shared libraries for extensions have always been. This change is intended to allow administrators to keep the Perl 5.004 library directory unchanged from a previous version, without running the risk of binary incompatibility between extensions' Perl source and shared libraries.

Module information summary

Brand new modules, arranged by topic rather than strictly alphabetically:

CGI.pm	Web server interface ("Common Gateway Interface")
CGI/Apache.pm	Support for Apache's Perl module
CGI/Carp.pm	Log server errors with helpful context
CGI/Fast.pm	Support for FastCGI (persistent server process)
CGI/Push.pm	Support for server push
CGI/Switch.pm	Simple interface for multiple server types
CPAN	Interface to Comprehensive Perl Archive Network
CPAN::FirstTime	Utility for creating CPAN configuration file
CPAN::Nox	Runs CPAN while avoiding compiled extensions
IO.pm	Top-level interface to IO::* classes
IO/File.pm	IO::File extension Perl module
IO/Handle.pm	IO::Handle extension Perl module
IO/Pipe.pm	IO::Pipe extension Perl module
IO/Seekable.pm	IO::Seekable extension Perl module
IO/Select.pm	IO::Select extension Perl module
IO/Socket.pm	IO::Socket extension Perl module
Opcode.pm	Disable named opcodes when compiling Perl code

ExtUtils/Embed.pm	Utilities for embedding Perl in C programs
ExtUtils/testlib.pm	Fixes up @INC to use just-built extension
FindBin.pm	Find path of currently executing program
Class/Struct.pm	Declare struct-like datatypes as Perl classes
File/stat.pm	By-name interface to Perl's builtin stat
Net/hostent.pm	By-name interface to Perl's builtin gethost*
Net/netent.pm	By-name interface to Perl's builtin getnet*
Net/protoent.pm	By-name interface to Perl's builtin getproto*
Net/servent.pm	By-name interface to Perl's builtin getserv*
Time/gmtime.pm	By-name interface to Perl's builtin gmtime
Time/localtime.pm	By-name interface to Perl's builtin localtime
Time/tm.pm	Internal object for Time::{gm,local}time
User/grent.pm	By-name interface to Perl's builtin getgr*
User/pwent.pm	By-name interface to Perl's builtin getpw*
Tie/RefHash.pm	Base class for tied hashes with references as keys
UNIVERSAL.pm	Base class for *ALL* classes

Fcntl

New constants in the existing Fcntl modules are now supported, provided that your operating system happens to support them:

```
F_GETOWN F_SETOWN
O_ASYNC O_DEFER O_DSYNC O_FSYNC O_SYNC
O_EXLOCK O_SHLOCK
```

These constants are intended for use with the Perl operators `sysopen()` and `fcntl()` and the basic database modules like `SDBM_File`. For the exact meaning of these and other Fcntl constants please refer to your operating system's documentation for `fcntl()` and `open()`.

In addition, the Fcntl module now provides these constants for use with the Perl operator `flock()`:

```
LOCK_SH LOCK_EX LOCK_NB LOCK_UN
```

These constants are defined in all environments (because where there is no `flock()` system call, Perl emulates it). However, for historical reasons, these constants are not exported unless they are explicitly requested with the `":flock"` tag (e.g. use `Fcntl ':flock'`).

IO

The IO module provides a simple mechanism to load all the IO modules at one go. Currently this includes:

```
IO::Handle
IO::Seekable
IO::File
IO::Pipe
IO::Socket
```

For more information on any of these modules, please see its respective documentation.

Math::Complex

The Math::Complex module has been totally rewritten, and now supports more operations. These are overloaded:

```
+ - * / ** <=> neg ~ abs sqrt exp log sin cos atan2 "" (stringify)
```

And these functions are now exported:

```
pi i Re Im arg
log10 logn ln cbrt root
```



```

tan
csc sec cot
asin acos atan
acsc asec acot
sinh cosh tanh
csch sech coth
asinh acosh atanh
acsch asech acoth
cplx cplx

```

Math::Trig

This new module provides a simpler interface to parts of Math::Complex for those who need trigonometric functions only for real numbers.

DB_File

There have been quite a few changes made to DB_File. Here are a few of the highlights:

- Fixed a handful of bugs.
- By public demand, added support for the standard hash function `exists()`.
- Made it compatible with Berkeley DB 1.86.
- Made negative subscripts work with RECNO interface.
- Changed the default flags from `O_RDWR` to `O_CREAT|O_RDWR` and the default mode from `0640` to `0666`.
- Made DB_File automatically import the `open()` constants (`O_RDWR`, `O_CREAT` etc.) from `Fcntl`, if available.
- Updated documentation.

Refer to the HISTORY section in DB_File.pm for a complete list of changes. Everything after DB_File 1.01 has been added since 5.003.

Net::Ping

Major rewrite – support added for both udp echo and real icmp pings.

Object-oriented overrides for builtin operators

Many of the Perl builtins returning lists now have object-oriented overrides. These are:

```

File::stat
Net::hostent
Net::netent
Net::protoent
Net::servent
Time::gmtime
Time::localtime
User::grent
User::pwent

```

For example, you can now say

```

use File::stat;
use User::pwent;
$this = (stat($filename)->st_uid == pwent($whoever)->pw_uid);

```

Utility Changes

pod2html

Sends converted HTML to standard output

The *pod2html* utility included with Perl 5.004 is entirely new. By default, it sends the converted HTML to its standard output, instead of writing it to a file like Perl 5.003's *pod2html* did. Use the **—outfile=FILENAME** option to write to a file.

xsubpp

`void` XSUBs now default to returning nothing

Due to a documentation/implementation bug in previous versions of Perl, XSUBs with a return type of `void` have actually been returning one value. Usually that value was the GV for the XSUB, but sometimes it was some already freed or reused value, which would sometimes lead to program failure.

In Perl 5.004, if an XSUB is declared as returning `void`, it actually returns no value, i.e. an empty list (though there is a backward-compatibility exception; see below). If your XSUB really does return an SV, you should give it a return type of `SV *`.

For backward compatibility, *xsubpp* tries to guess whether a `void` XSUB is really `void` or if it wants to return an `SV *`. It does so by examining the text of the XSUB: if *xsubpp* finds what looks like an assignment to `ST(0)`, it assumes that the XSUB's return type is really `SV *`.

C Language API Changes

`gv_fetchmethod` and `perl_call_sv`

The `gv_fetchmethod` function finds a method for an object, just like in Perl 5.003. The GV it returns may be a method cache entry. However, in Perl 5.004, method cache entries are not visible to users; therefore, they can no longer be passed directly to `perl_call_sv`. Instead, you should use the `GvCV` macro on the GV to extract its CV, and pass the CV to `perl_call_sv`.

The most likely symptom of passing the result of `gv_fetchmethod` to `perl_call_sv` is Perl's producing an "Undefined subroutine called" error on the *second* call to a given method (since there is no cache on the first call).

`perl_eval_pv`

A new function handy for eval'ing strings of Perl code inside C code. This function returns the value from the eval statement, which can be used instead of fetching globals from the symbol table. See [perlguts](#), [perlembed](#) and [perlcalls](#) for details and examples.

Extended API for manipulating hashes

Internal handling of hash keys has changed. The old hashtable API is still fully supported, and will likely remain so. The additions to the API allow passing keys as `SV*`s, so that `tied` hashes can be given real scalars as keys rather than plain strings (nontied hashes still can only use strings as keys). New extensions must use the new hash access functions and macros if they wish to use `SV*` keys. These additions also make it feasible to manipulate `HE*`s (hash entries), which can be more efficient. See [perlguts](#) for details.

Documentation Changes

Many of the base and library pods were updated. These new pods are included in section 1:

[perldelta](#)

This document.

[perlfaq](#)

Frequently asked questions.

[perllocale](#)

Locale support (internationalization and localization).

[perltoot](#)

Tutorial on Perl OO programming.

perlapi

Perl internal IO abstraction interface.

perlmodlib

Perl module library and recommended practice for module creation. Extracted from *perlmod* (which is much smaller as a result).

perldebug

Although not new, this has been massively updated.

perlsec

Although not new, this has been massively updated.

New Diagnostics

Several new conditions will trigger warnings that were silent before. Some only affect certain platforms. The following new warnings and errors outline these. These messages are classified as follows (listed in increasing order of desperation):

- (W) A warning (optional).
- (D) A deprecation (optional).
- (S) A severe warning (mandatory).
- (F) A fatal error (trappable).
- (P) An internal error you should never see (trappable).
- (X) A very fatal error (nontrappable).
- (A) An alien error message (not generated by Perl).

"my" variable %s masks earlier declaration in same scope

(W) A lexical variable has been redeclared in the same scope, effectively eliminating all access to the previous instance. This is almost always a typographical error. Note that the earlier variable will still exist until the end of the scope or until all closure referents to it are destroyed.

%s argument is not a HASH element or slice

(F) The argument to `delete()` must be either a hash element, such as

```
$foo{$bar}
$ref->[12]->{"susie"}
```

or a hash slice, such as

```
@foo{$bar, $baz, $xyzyzy}
@{$ref->[12]}{"susie", "queue"}
```

Allocation too large: %lx

(X) You can't allocate more than 64K on an MS-DOS machine.

Allocation too large

(F) You can't allocate more than $2^{31} + \text{"small amount"}$ bytes.

Applying %s to %s will act on scalar(%s)

(W) The pattern match (`/`), substitution (`s///`), and transliteration (`tr///`) operators work on scalar values. If you apply one of them to an array or a hash, it will convert the array or hash to a scalar value — the length of an array, or the population info of a hash — and then work on that scalar value. This is probably not what you meant to do. See *grep* and *map* for alternatives.

Attempt to free nonexistent shared string

(P) Perl maintains a reference counted internal table of strings to optimize the storage and access of hash keys and other strings. This indicates someone tried to decrement the reference count of a string that can no longer be found in the table.

Attempt to use reference as lvalue in substr

(W) You supplied a reference as the first argument to `substr()` used as an lvalue, which is pretty strange. Perhaps you forgot to dereference it first. See [substr](#).

Bareword "%s" refers to nonexistent package

(W) You used a qualified bareword of the form `Foo::`, but the compiler saw no other uses of that namespace before that point. Perhaps you need to predeclare a package?

Can't redefine active sort subroutine %s

(F) Perl optimizes the internal handling of sort subroutines and keeps pointers into them. You tried to redefine one such sort subroutine when it was currently active, which is not allowed. If you really want to do this, you should write `sort { &func } @x` instead of `sort func @x`.

Can't use bareword ("%s") as %s ref while "strict refs" in use

(F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See [perlref](#).

Cannot resolve method '%s' overloading '%s' in package '%s'

(P) Internal error trying to resolve overloading specified by a method name (as opposed to a subroutine reference).

Constant subroutine %s redefined

(S) You redefined a subroutine which had previously been eligible for inlining. See [Constant Functions in perlsyn](#) for commentary and workarounds.

Constant subroutine %s undefined

(S) You undefined a subroutine which had previously been eligible for inlining. See [Constant Functions in perlsyn](#) for commentary and workarounds.

Copy method did not return a reference

(F) The method which overloads "=" is buggy. See [Copy Constructor](#).

Died

(F) You passed `die()` an empty string (the equivalent of `die ""`) or you called it with no args and both `$@` and `$_` were empty.

Exiting pseudo-block via %s

(W) You are exiting a rather special block construct (like a sort block or subroutine) by unconventional means, such as a `goto`, or a loop control statement. See [sort](#).

Identifier too long

(F) Perl limits identifiers (names for variables, functions, etc.) to 252 characters for simple names, somewhat more for compound names (like `$A::B`). You've exceeded Perl's limits. Future versions of Perl are likely to eliminate these arbitrary limitations.

Illegal character %s (carriage return)

(F) A carriage return character was found in the input. This is an error, and not a warning, because carriage return characters can break multi-line strings, including here documents (e.g., `print <<EOF;`).

Illegal switch in PERL5OPT: %s

(X) The `PERL5OPT` environment variable may only be used to set the following switches: `-[DIMUdmw]`.

Integer overflow in hex number

(S) The literal hex number you have specified is too big for your architecture. On a 32-bit architecture the largest hex literal is `0xFFFFFFFF`.

Integer overflow in octal number

(S) The literal octal number you have specified is too big for your architecture. On a 32-bit architecture the largest octal literal is 037777777777.

internal error: glob failed

(P) Something went wrong with the external program(s) used for `glob` and `<*.c`. This may mean that your `csh` (C shell) is broken. If so, you should change all of the `csh`-related variables in `config.sh`: If you have `tcsh`, make the variables refer to it as if it were `csh` (e.g. `full_csh='/usr/bin/tcsh'`); otherwise, make them all empty (except that `d_csh` should be `'undef'`) so that Perl will think `csh` is missing. In either case, after editing `config.sh`, run `./Configure -S` and rebuild Perl.

Invalid conversion in %s: "%s"

(W) Perl does not understand the given format conversion. See [sprintf](#).

Invalid type in pack: '%s'

(F) The given character is not a valid pack type. See [pack](#).

Invalid type in unpack: '%s'

(F) The given character is not a valid unpack type. See [unpack](#).

Name "%s::%s" used only once: possible typo

(W) Typographical errors often show up as unique variable names. If you had a good reason for having a unique name, then just mention it again somehow to suppress the message (the `use vars` pragma is provided for just this purpose).

Null picture in formline

(F) The first argument to `formline` must be a valid format picture specification. It was found to be empty, which probably means you supplied it an uninitialized value. See [perlform](#).

Offset outside string

(F) You tried to do a `read/write/send/rcv` operation with an offset pointing outside the buffer. This is difficult to imagine. The sole exception to this is that `sysread()` ing past the buffer will extend the buffer and zero pad the new area.

Out of memory!

(X|F) The `malloc()` function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request.

The request was judged to be small, so the possibility to trap it depends on the way Perl was compiled. By default it is not trappable. However, if compiled for this, Perl may use the contents of `$_M` as an emergency pool after `die()` ing with this message. In this case the error is trappable *once*.

Out of memory during request for %s

(F) The `malloc()` function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request. However, the request was judged large enough (compile-time default is 64K), so a possibility to shut down by trapping this error is granted.

panic: frexp

(P) The library function `frexp()` failed, making `printf("%f")` impossible.

Possible attempt to put comments in qw() list

(W) `qw()` lists contain items separated by whitespace; as with literal strings, comment characters are not ignored, but are instead treated as literal data. (You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
@list = qw(
    a # a comment
    b # another comment
);
```

when you should have written this:

```
@list = qw(
    a
    b
);
```

If you really want comments, build your list the old-fashioned way, with quotes and commas:

```
@list = (
    'a',      # a comment
    'b',      # another comment
);
```

Possible attempt to separate words with commas

(W) `qw()` lists contain items separated by whitespace; therefore commas aren't needed to separate the items. (You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
qw! a, b, c !;
```

which puts literal commas into some of the list items. Write it without commas if you don't want them to appear in your data:

```
qw! a b c !;
```

Scalar value `@%s{%s}` better written as `$_s{%s}`

(W) You've used a hash slice (indicated by `@`) to select a single element of a hash. Generally it's better to ask for a scalar value (indicated by `$`). The difference is that `$_foo{&bar}` always behaves like a scalar, both when assigning to it and when evaluating its argument, while `@foo{&bar}` behaves like a list when you assign to it, and provides a list context to its subscript, which can do weird things if you're expecting only one subscript.

Stub found while resolving method '`%s`' overloading '`%s`' in `%s`

(P) Overloading resolution over `@ISA` tree may be broken by importing stubs. Stubs should never be implicitly created, but explicit calls to `can` may break this.

Too late for "`-T`" option

(X) The `#!` line (or local equivalent) in a Perl script contains the `-T` option, but Perl was not invoked with `-T` in its argument list. This is an error because, by the time Perl discovers a `-T` in a script, it's too late to properly taint everything from the environment. So Perl gives up.

`untie` attempted while `%d` inner references still exist

(W) A copy of the object returned from `tie` (or `tied`) was still valid when `untie` was called.

Unrecognized character `%s`

(F) The Perl parser has no idea what to do with the specified character in your Perl script (or eval). Perhaps you tried to run a compressed script, a binary program, or a directory as a Perl program.

Unsupported function `fork`

(F) Your version of executable does not support forking.

Note that under some systems, like OS/2, there may be different flavors of Perl executables, some of which may support `fork`, some not. Try changing the name you call Perl by to `perl_`, `perl__`, and

so on.

Use of "\$\$<digit" to mean "\${}\$<digit" is deprecated

(D) Perl versions before 5.004 misinterpreted any type marker followed by "\$" and a digit. For example, "\$\$0" was incorrectly taken to mean "\${}\$0" instead of "\${\$0}". This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "\$\$0" in a string. So Perl 5.004 still interprets "\$\$<digit" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

Value of %s can be "0"; test with defined()

(W) In a conditional expression, you used <HANDLE, <*(glob), each(), or readdir() as a boolean value. Each of these constructs can return a value of "0"; that would make the conditional expression false, which is probably not what you intended. When using these constructs in conditional expressions, test their values with the defined operator.

Variable "%s" may be unavailable

(W) An inner (nested) *anonymous* subroutine is inside a *named* subroutine, and outside that is another subroutine; and the anonymous (innermost) subroutine is referencing a lexical variable defined in the outermost subroutine. For example:

```
sub outermost { my $a; sub middle { sub { $a } } }
```

If the anonymous subroutine is called or referenced (directly or indirectly) from the outermost subroutine, it will share the variable as you would expect. But if the anonymous subroutine is called or referenced when the outermost subroutine is not active, it will see the value of the shared variable as it was before and during the **first** call to the outermost subroutine, which is probably not what you want.

In these circumstances, it is usually best to make the middle subroutine anonymous, using the sub {} syntax. Perl has specific support for shared variables in nested anonymous subroutines; a named subroutine in between interferes with this feature.

Variable "%s" will not stay shared

(W) An inner (nested) *named* subroutine is referencing a lexical variable defined in an outer subroutine.

When the inner subroutine is called, it will probably see the value of the outer subroutine's variable as it was before and during the **first** call to the outer subroutine; in this case, after the first call to the outer subroutine is complete, the inner and outer subroutines will no longer share a common value for the variable. In other words, the variable will no longer be shared.

Furthermore, if the outer subroutine is anonymous and references a lexical variable outside itself, then the outer and inner subroutines will *never* share the given variable.

This problem can usually be solved by making the inner subroutine anonymous, using the sub {} syntax. When inner anonymous subs that reference variables in outer subroutines are called or referenced, they are automatically rebound to the current values of such variables.

Warning: something's wrong

(W) You passed warn() an empty string (the equivalent of warn "") or you called it with no args and \$_ was empty.

Ill-formed logical name [%s] in prime_env_iter

(W) A warning peculiar to VMS. A logical name was encountered when preparing to iterate over %ENV which violates the syntactic rules governing logical names. Since it cannot be translated normally, it is skipped, and will not appear in %ENV. This may be a benign occurrence, as some software packages might directly modify logical name tables and introduce nonstandard names, or it

may indicate that a logical name table has been corrupted.

Got an error from DosAllocMem

(P) An error peculiar to OS/2. Most probably you're using an obsolete version of Perl, and this should not happen anyway.

Malformed PERLLIB_PREFIX

(F) An error peculiar to OS/2. PERLLIB_PREFIX should be of the form

```
prefix1;prefix2
```

or

```
prefix1 prefix2
```

with nonempty prefix1 and prefix2. If prefix1 is indeed a prefix of a builtin library search path, prefix2 is substituted. The error may appear if components are not found, or are too long. See "PERLLIB_PREFIX" in *README.os2*.

PERL_SH_DIR too long

(F) An error peculiar to OS/2. PERL_SH_DIR is the directory to find the sh-shell in. See "PERL_SH_DIR" in *README.os2*.

Process terminated by SIG%s

(W) This is a standard message issued by OS/2 applications, while *nix applications die in silence. It is considered a feature of the OS/2 port. One can easily disable this by appropriate sighandlers, see *Signals in perlpc*. See also "Process terminated by SIGTERM/SIGINT" in *README.os2*.

BUGS

If you find what you think is a bug, you might check the headers of recently posted articles in the comp.lang.perl.misc newsgroup. There may also be information at <http://www.perl.com/perl/>, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Make sure you trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to [<perlbug@perl.com>](mailto:perlbug@perl.com) to be analysed by the Perl porting team.

SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl. This file has been significantly updated for 5.004, so even veteran users should look through it.

The *README* file for general stuff.

The *Copying* file for copyright information.

HISTORY

Constructed by Tom Christiansen, grabbing material with permission from innumerable contributors, with kibitzing by more than a few Perl porters.

Last update: Wed May 14 11:14:09 EDT 1997

NAME

perl5005delta – what's new for perl5.005

DESCRIPTION

This document describes differences between the 5.004 release and this one.

About the new versioning system

Perl is now developed on two tracks: a maintenance track that makes small, safe updates to released production versions with emphasis on compatibility; and a development track that pursues more aggressive evolution. Maintenance releases (which should be considered production quality) have subversion numbers that run from 1 to 49, and development releases (which should be considered "alpha" quality) run from 50 to 99.

Perl 5.005 is the combined product of the new dual-track development scheme.

Incompatible Changes

WARNING: This version is not binary compatible with Perl 5.004.

Starting with Perl 5.004_50 there were many deep and far-reaching changes to the language internals. If you have dynamically loaded extensions that you built under perl 5.003 or 5.004, you can continue to use them with 5.004, but you will need to rebuild and reinstall those extensions to use them 5.005. See *INSTALL* for detailed instructions on how to upgrade.

Default installation structure has changed

The new Configure defaults are designed to allow a smooth upgrade from 5.004 to 5.005, but you should read *INSTALL* for a detailed discussion of the changes in order to adapt them to your system.

Perl Source Compatibility

When none of the experimental features are enabled, there should be very few user-visible Perl source compatibility issues.

If threads are enabled, then some caveats apply. @_ and \$_ become lexical variables. The effect of this should be largely transparent to the user, but there are some boundary conditions under which user will need to be aware of the issues. For example, `local(@_)` results in a "Can't localize lexical variable @_ ..." message. This may be enabled in a future version.

Some new keywords have been introduced. These are generally expected to have very little impact on compatibility. See *New INIT keyword*, *New lock keyword*, and *New qr operator*.

Certain barewords are now reserved. Use of these will provoke a warning if you have asked for them with the -w switch. See *our is now a reserved word*.

C Source Compatibility

There have been a large number of changes in the internals to support the new features in this release.

- Core sources now require ANSI C compiler

An ANSI C compiler is now **required** to build perl. See *INSTALL*.

- All Perl global variables must now be referenced with an explicit prefix

All Perl global variables that are visible for use by extensions now have a PL_ prefix. New extensions should not refer to perl globals by their unqualified names. To preserve sanity, we provide limited backward compatibility for globals that are being widely used like `sv_undef` and `na` (which should now be written as `PL_sv_undef`, `PL_na` etc.)

If you find that your XS extension does not compile anymore because a perl global is not visible, try adding a PL_ prefix to the global and rebuild.

It is strongly recommended that all functions in the Perl API that don't begin with `perl` be referenced with a `Perl_` prefix. The bare function names without the `Perl_` prefix are supported with macros,

but this support may cease in a future release.

See [API LISTING in perl guts](#).

- Enabling threads has source compatibility issues

Perl built with threading enabled requires extensions to use the new `dTHR` macro to initialize the handle to access per-thread data. If you see a compiler error that talks about the variable `thr` not being declared (when building a module that has XS code), you need to add `dTHR;` at the beginning of the block that elicited the error.

The API function `perl_get_sv("@", FALSE)` should be used instead of directly accessing perl globals as `GvSV(errgv)`. The API call is backward compatible with existing perls and provides source compatibility with threading is enabled.

See ["C Source Compatibility"](#) for more information.

Binary Compatibility

This version is NOT binary compatible with older versions. All extensions will need to be recompiled. Further binaries built with threads enabled are incompatible with binaries built without. This should largely be transparent to the user, as all binary incompatible configurations have their own unique architecture name, and extension binaries get installed at unique locations. This allows coexistence of several configurations in the same directory hierarchy. See *INSTALL*.

Security fixes may affect compatibility

A few taint leaks and taint omissions have been corrected. This may lead to "failure" of scripts that used to work with older versions. Compiling with `-DINCOMPLETE_TAINTS` provides a perl with minimal amounts of changes to the tainting behavior. But note that the resulting perl will have known insecurities.

Oneliners with the `-e` switch do not create temporary files anymore.

Relaxed new mandatory warnings introduced in 5.004

Many new warnings that were introduced in 5.004 have been made optional. Some of these warnings are still present, but perl's new features make them less often a problem. See [New Diagnostics](#).

Licensing

Perl has a new Social Contract for contributors. See *Porting/Contract*.

The license included in much of the Perl documentation has changed. Most of the Perl documentation was previously under the implicit GNU General Public License or the Artistic License (at the user's choice). Now much of the documentation unambiguously states the terms under which it may be distributed. Those terms are in general much less restrictive than the GNU GPL. See [perl](#) and the individual perl manpages listed therein.

Core Changes

Threads

WARNING: Threading is considered an **experimental** feature. Details of the implementation may change without notice. There are known limitations and some bugs. These are expected to be fixed in future versions.

See *README.threads*.

Compiler

WARNING: The Compiler and related tools are considered **experimental**. Features may change without notice, and there are known limitations and bugs. Since the compiler is fully external to perl, the default configuration will build and install it.

The Compiler produces three different types of transformations of a perl program. The C backend generates C code that captures perl's state just before execution begins. It eliminates the compile-time overheads of the regular perl interpreter, but the run-time performance remains comparatively the same. The CC backend

generates optimized C code equivalent to the code path at run-time. The CC backend has greater potential for big optimizations, but only a few optimizations are implemented currently. The Bytecode backend generates a platform independent bytecode representation of the interpreter's state just before execution. Thus, the Bytecode back end also eliminates much of the compilation overhead of the interpreter.

The compiler comes with several valuable utilities.

`B::Lint` is an experimental module to detect and warn about suspicious code, especially the cases that the `-w` switch does not detect.

`B::Deparse` can be used to demystify perl code, and understand how perl optimizes certain constructs.

`B::Xref` generates cross reference reports of all definition and use of variables, subroutines and formats in a program.

`B::Showlex` show the lexical variables used by a subroutine or file at a glance.

`perlcc` is a simple frontend for compiling perl.

See `ext/B/README`, [B](#), and the respective compiler modules.

Regular Expressions

Perl's regular expression engine has been seriously overhauled, and many new constructs are supported. Several bugs have been fixed.

Here is an itemized summary:

Many new and improved optimizations

Changes in the RE engine:

- Unneeded nodes removed;
- Substrings merged together;
- New types of nodes to process `(SUBEXPR)*` and similar expressions quickly, used if the SUBEXPR has no side effects and matches strings of the same length;
- Better optimizations by lookup for constant substrings;
- Better search for constants substrings anchored by `$` ;

Changes in Perl code using RE engine:

- More optimizations to `s/longer/short/`;
- `study()` was not working;
- `/blah/` may be optimized to an analogue of `index()` if `$&` `$'` `$'` not seen;
- Unneeded copying of matched-against string removed;
- Only matched part of the string is copying if `$'` `$'` were not seen;

Many bug fixes

Note that only the major bug fixes are listed here. See *Changes* for others.

- Backtracking might not restore start of `$3`.
- No feedback if max count for `*` or `+` on "complex" subexpression was reached, similarly (but at compile time) for `{3,34567}`
- Primitive restrictions on max count introduced to decrease a possibility of a segfault;
- `(ZERO-LENGTH)*` could segfault;
- `(ZERO-LENGTH)*` was prohibited;
- Long REs were not allowed;
- `/RE/g` could skip matches at the same position after a zero-length match;

New regular expression constructs

The following new syntax elements are supported:

```
(?<=RE)
(?<!RE)
(?{ CODE })
(?i-x)
(?i:RE)
(? (COND) YES_RE|NO_RE)
(?>RE)
\z
```

New operator for precompiled regular expressions

See [New `qr` operator](#).

Other improvements

Better debugging output (possibly with colors),
 even from non-debugging Perl;
 RE engine code now looks like C, not like assembler;
 Behaviour of RE modifiable by 'use re' directive;
 Improved documentation;
 Test suite significantly extended;
 Syntax [:^(upper:)] etc., reserved inside character classes;

Incompatible changes

```
(?i) localized inside enclosing group;
$( is not interpolated into RE any more;
/RE/g may match at the same position (with non-zero length)
after a zero-length match (bug fix).
```

See [perlre](#) and [perlop](#).

Improved `malloc()`

See banner at the beginning of `malloc.c` for details.

Quicksort is internally implemented

Perl now contains its own highly optimized `qsort()` routine. The new `qsort()` is resistant to inconsistent comparison functions, so Perl's `sort()` will not provoke coredumps any more when given poorly written sort subroutines. (Some C library `qsort()`s that were being used before used to have this problem.) In our testing, the new `qsort()` required the minimal number of pair-wise compares on average, among all known `qsort()` implementations.

See `perlfunc/sort`.

Reliable signals

Perl's signal handling is susceptible to random crashes, because signals arrive asynchronously, and the Perl runtime is not reentrant at arbitrary times.

However, one experimental implementation of reliable signals is available when threads are enabled. See `Thread::Signal`. Also see **INSTALL** for how to build a Perl capable of threads.

Reliable stack pointers

The internals now reallocate the perl stack only at predictable times. In particular, magic calls never trigger reallocations of the stack, because all reentrancy of the runtime is handled using a "stack of stacks". This should improve reliability of cached stack pointers in the internals and in XSUBS.

More generous treatment of carriage returns

Perl used to complain if it encountered literal carriage returns in scripts. Now they are mostly treated like whitespace within program text. Inside string literals and here documents, literal carriage returns are ignored if they occur paired with linefeeds, or get interpreted as whitespace if they stand alone. This behavior means that literal carriage returns in files should be avoided. You can get the older, more compatible (but less generous) behavior by defining the preprocessor symbol `PERL_STRICT_CR` when building perl. Of course, all this has nothing whatever to do with how escapes like `\r` are handled within strings.

Note that this doesn't somehow magically allow you to keep all text files in DOS format. The generous treatment only applies to files that perl itself parses. If your C compiler doesn't allow carriage returns in files, you may still be unable to build modules that need a C compiler.

Memory leaks

`substr`, `pos` and `vec` don't leak memory anymore when used in lvalue context. Many small leaks that impacted applications that embed multiple interpreters have been fixed.

Better support for multiple interpreters

The build-time option `-DMULTIPLICITY` has had many of the details reworked. Some previously global variables that should have been per-interpreter now are. With care, this allows interpreters to call each other. See the `PerlInterp` extension on CPAN.

Behavior of `local()` on array and hash elements is now well-defined

See *"Temporary Values via `local()`"*.

`%!` is transparently tied to the *Errno* module

See *perlvar*, and *Errno*.

Pseudo-hashes are supported

See *perlref*.

`EXPR foreach EXPR` is supported

See *perlsyn*.

Keywords can be globally overridden

See *perlsub*.

`$^E` is meaningful on Win32

See *perlvar*.

`foreach (1..1000000)` optimized

`foreach (1..1000000)` is now optimized into a counting loop. It does not try to allocate a 1000000-size list anymore.

`Foo::` can be used as implicitly quoted package name

Barewords caused unintuitive behavior when a subroutine with the same name as a package happened to be defined. Thus, `new Foo @args`, use the result of the call to `Foo()` instead of `Foo` being treated as a literal. The recommended way to write barewords in the indirect object slot is `new Foo:: @args`. Note that the method `new()` is called with a first argument of `Foo`, not `Foo::` when you do that.

`exists $Foo::{Bar::}` tests existence of a package

It was impossible to test for the existence of a package without actually creating it before. Now `exists $Foo::{Bar::}` can be used to test if the `Foo::Bar` namespace has been created.

Better locale support

See *perllocale*.

Experimental support for 64-bit platforms

Perl5 has always had 64-bit support on systems with 64-bit longs. Starting with 5.005, the beginnings of experimental support for systems with 32-bit long and 64-bit 'long long' integers has been added. If you add `-DUSE_LONG_LONG` to your `ccflags` in `config.sh` (or manually define it in `perl.h`) then perl will be built with 'long long' support. There will be many compiler warnings, and the resultant perl may not work on all systems. There are many other issues related to third-party extensions and libraries. This option exists to allow people to work on those issues.

`prototype()` returns useful results on builtins

See [prototype](#).

Extended support for exception handling

`die()` now accepts a reference value, and `$@` gets set to that value in exception traps. This makes it possible to propagate exception objects. This is an undocumented **experimental** feature.

Re-blessing in `DESTROY()` supported for chaining `DESTROY()` methods

See [Destructors](#).

All `printf` format conversions are handled internally

See [printf](#).

New `INIT` keyword

`INIT` subs are like `BEGIN` and `END`, but they get run just before the perl runtime begins execution. e.g., the Perl Compiler makes use of `INIT` blocks to initialize and resolve pointers to `XSUBs`.

New `lock` keyword

The `lock` keyword is the fundamental synchronization primitive in threaded perl. When threads are not enabled, it is currently a noop.

To minimize impact on source compatibility this keyword is "weak", i.e., any user-defined subroutine of the same name overrides it, unless a `use Thread` has been seen.

New `qr//` operator

The `qr//` operator, which is syntactically similar to the other quote-like operators, is used to create precompiled regular expressions. This compiled form can now be explicitly passed around in variables, and interpolated in other regular expressions. See [perlop](#).

`our` is now a reserved word

Calling a subroutine with the name `our` will now provoke a warning when using the `-w` switch.

Tied arrays are now fully supported

See [Tie::Array](#).

Tied handles support is better

Several missing hooks have been added. There is also a new base class for `TIEARRAY` implementations. See [Tie::Array](#).

4th argument to `substr`

`substr()` can now both return and replace in one operation. The optional 4th argument is the replacement string. See [substr](#).

Negative `LENGTH` argument to `splice`

`splice()` with a negative `LENGTH` argument now work similar to what the `LENGTH` did for `substr()`. Previously a negative `LENGTH` was treated as 0. See [splice](#).

Magic lvalues are now more magical

When you say something like `substr($x, 5) = "hi"`, the scalar returned by `substr()` is special, in that any modifications to it affect `$x`. (This is called a 'magic lvalue' because an 'lvalue' is something on

the left side of an assignment.) Normally, this is exactly what you would expect to happen, but Perl uses the same magic if you use `substr()`, `pos()`, or `vec()` in a context where they might be modified, like taking a reference with `\` or as an argument to a sub that modifies `@_`. In previous versions, this ‘magic’ only went one way, but now changes to the scalar the magic refers to (`$x` in the above example) affect the magic lvalue too. For instance, this code now acts differently:

```
$x = "hello";
sub printit {
    $x = "g'bye";
    print $_[0], "\n";
}
printit(substr($x, 0, 5));
```

In previous versions, this would print "hello", but it now prints "g'bye".

< now reads in records

If `$/` is a reference to an integer, or a scalar that holds an integer, `<` will read in records instead of lines. For more info, see [\\$/.](#)

Supported Platforms

Configure has many incremental improvements. Site-wide policy for building perl can now be made persistent, via `Policy.sh`. Configure also records the command-line arguments used in *config.sh*.

New Platforms

BeOS is now supported. See *README.beos*.

DOS is now supported under the DJGPP tools. See *README.dos* (installed as [perldos](#) on some systems).

MiNT is now supported. See *README.mint*.

MPE/iX is now supported. See *README.mpeix*.

MVS (aka OS390, aka Open Edition) is now supported. See *README.os390* (installed as [perlos390](#) on some systems).

Stratus VOS is now supported. See *README.vos*.

Changes in existing support

Win32 support has been vastly enhanced. Support for Perl Object, a C++ encapsulation of Perl. GCC and EGCS are now supported on Win32. See *README.win32*, aka [perlwin32](#).

VMS configuration system has been rewritten. See *README.vms* (installed as [README_vms](#) on some systems).

The hints files for most Unix platforms have seen incremental improvements.

Modules and Pragmata

New Modules

B Perl compiler and tools. See [B](#).

Data::Dumper

A module to pretty print Perl data. See [Data::Dumper](#).

Dumpvalue

A module to dump perl values to the screen. See [Dumpvalue](#).

Errno

A module to look up errors more conveniently. See [Errno](#).

File::Spec

A portable API for file operations.

ExtUtils::Installed

Query and manage installed modules.

ExtUtils::Packlist

Manipulate .packlist files.

Fatal

Make functions/builtins succeed or die.

IPC::SysV

Constants and other support infrastructure for System V IPC operations in perl.

Test

A framework for writing testsuites.

Tie::Array

Base class for tied arrays.

Tie::Handle

Base class for tied handles.

Thread

Perl thread creation, manipulation, and support.

attrs

Set subroutine attributes.

fields

Compile-time class fields.

re Various pragmata to control behavior of regular expressions.

Changes in existing modules**Benchmark**

You can now run tests for *x* seconds instead of guessing the right number of tests to run.

Keeps better time.

Carp

Carp has a new function `cluck()`. `cluck()` warns, like `carp()`, but also adds a stack backtrace to the error message, like `confess()`.

CGI CGI has been updated to version 2.42.

Fcntl

More Fcntl constants added: `F_SETLK64`, `F_SETLKW64`, `O_LARGEFILE` for large (more than 4G) file access (the 64-bit support is not yet working, though, so no need to get overly excited), Free/Net/OpenBSD locking behaviour flags `F_FLOCK`, `F_POSIX`, Linux `F_SHLCK`, and `O_ACCMODE`: the mask of `O_RDONLY`, `O_WRONLY`, and `O_RDWR`.

Math::Complex

The accessor methods `Re`, `Im`, `arg`, `abs`, `rho`, `theta`, methods `can ($z-Re())` now also act as mutators (`$z-Re(3)`).

Math::Trig

A little bit of radial trigonometry (cylindrical and spherical) added, for example the great circle distance.

POSIX

POSIX now has its own platform-specific hints files.

DB_File

DB_File supports version 2.x of Berkeley DB. See `ext/DB_File/Changes`.

MakeMaker

MakeMaker now supports writing empty makefiles, provides a way to specify that `umask()` policy should be honored. There is also better support for manipulation of `.packlist` files, and getting information about installed modules.

Extensions that have both architecture-dependent and architecture-independent files are now always installed completely in the architecture-dependent locations. Previously, the shareable parts were shared both across architectures and across perl versions and were therefore liable to be overwritten with newer versions that might have subtle incompatibilities.

CPAN

See [perlmodinstall](#) and [CPAN](#).

Cwd

`Cwd::cwd` is faster on most platforms.

Utility Changes

`h2ph` and related utilities have been vastly overhauled.

`perlcc`, a new experimental front end for the compiler is available.

The crude GNU `configure` emulator is now called `configure.gnu` to avoid trampling on `Configure` under case-insensitive filesystems.

`perldoc` used to be rather slow. The slower features are now optional. In particular, case-insensitive searches need the `-i` switch, and recursive searches need `-r`. You can set these switches in the `PERLDOC` environment variable to get the old behavior.

Documentation Changes

`Config.pm` now has a glossary of variables.

[Porting/patching.pod](#) has detailed instructions on how to create and submit patches for perl.

[perlport](#) specifies guidelines on how to write portably.

[perlmodinstall](#) describes how to fetch and install modules from CPAN sites.

Some more Perl traps are documented now. See [perltrap](#).

[perlopentut](#) gives a tutorial on using `open()`.

[perlreftut](#) gives a tutorial on references.

[perlthrtut](#) gives a tutorial on threads.

New Diagnostics

Ambiguous call resolved as `CORE::%s()`, qualify as such or use `&`

(W) A subroutine you have declared has the same name as a Perl keyword, and you have used the name without qualification for calling one or the other. Perl decided to call the builtin because the subroutine is not imported.

To force interpretation as a subroutine call, either put an ampersand before the subroutine name, or qualify the name with its package. Alternatively, you can import the subroutine (or pretend that it's imported with the `use subs pragma`).

To silently interpret it as the Perl operator, use the `CORE::` prefix on the operator (e.g. `CORE::log($x)`) or by declaring the subroutine to be an object method (see [attrs](#)).

Bad index while coercing array into hash

(F) The index looked up in the hash found as the 0'th element of a pseudo-hash is not legal. Index values must be at 1 or greater. See [perlref](#).

Bareword "%s" refers to nonexistent package

(W) You used a qualified bareword of the form `Foo::`, but the compiler saw no other uses of that namespace before that point. Perhaps you need to predeclare a package?

Can't call method "%s" on an undefined value

(F) You used the syntax of a method call, but the slot filled by the object reference or package name contains an undefined value. Something like this will reproduce the error:

```
$BADREF = 42;
process $BADREF 1, 2, 3;
$BADREF->process(1, 2, 3);
```

Can't check filesystem of script "%s" for nosuid

(P) For some reason you can't check the filesystem of the script for nosuid.

Can't coerce array into hash

(F) You used an array where a hash was expected, but the array has no information on how to map from keys to array indices. You can do that only with arrays that have a hash reference at index 0.

Can't goto subroutine from an eval-string

(F) The "goto subroutine" call can't be used to jump out of an eval "string". (You can use it to jump out of an eval {BLOCK}, but you probably don't want to.)

Can't localize pseudo-hash element

(F) You said something like `< local $ar-{ 'key' }`, where `$ar` is a reference to a pseudo-hash. That hasn't been implemented yet, but you can get a similar effect by localizing the corresponding array element directly — `< local $ar-[$ar-[0]{'key'}]`.

Can't use %%! because Errno.pm is not available

(F) The first time the %! hash is used, perl automatically loads the Errno.pm module. The Errno module is expected to tie the %! hash to provide symbolic names for \$! errno values.

Cannot find an opnumber for "%s"

(F) A string of a form `CORE::word` was given to `prototype()`, but there is no builtin with the name `word`.

Character class syntax `[. .]` is reserved for future extensions

(W) Within regular expression character classes (`[]`) the syntax beginning with `"["` and ending with `"]"` is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: `"\[."` and `"]"`.

Character class syntax `[: :]` is reserved for future extensions

(W) Within regular expression character classes (`[]`) the syntax beginning with `":["` and ending with `"]"` is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: `"\[:"` and `"]"`.

Character class syntax [= =] is reserved for future extensions

(W) Within regular expression character classes ([]) the syntax beginning with "[=" and ending with "]" is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: "\[" and "\]".

%s: Eval-group in insecure regular expression

(F) Perl detected tainted data when trying to compile a regular expression that contains the (?{ . . . }) zero-width assertion, which is unsafe. See (*{ code }*), and *perlsec*.

%s: Eval-group not allowed, use re 'eval'

(F) A regular expression contained the (?{ . . . }) zero-width assertion, but that construct is only allowed when the use re 'eval' pragma is in effect. See (*{ code }*).

%s: Eval-group not allowed at run time

(F) Perl tried to compile a regular expression containing the (?{ . . . }) zero-width assertion at run time, as it would when the pattern contains interpolated values. Since that is a security risk, it is not allowed. If you insist, you may still do this by explicitly building the pattern from an interpolated string at run time and using that in an eval(). See (*{ code }*).

Explicit blessing to "" (assuming package main)

(W) You are blessing a reference to a zero length string. This has the effect of blessing the reference into the package main. This is usually not what you want. Consider providing a default target package, e.g. bless(\$ref, \$p || 'MyPackage');

Illegal hex digit ignored

(W) You may have tried to use a character other than 0 – 9 or A – F in a hexadecimal number. Interpretation of the hexadecimal number stopped before the illegal character.

No such array field

(F) You tried to access an array as a hash, but the field name used is not defined. The hash at index 0 should map all valid field names to array indices for that to work.

No such field "%s" in variable %s of type %s

(F) You tried to access a field of a typed variable where the type does not know about the field name. The field names are looked up in the %FIELDS hash in the type package at compile time. The %FIELDS hash is usually set up with the 'fields' pragma.

Out of memory during ridiculously large request

(F) You can't allocate more than 2³¹+ "small amount" bytes. This error is most likely to be caused by a typo in the Perl program. e.g., \$arr[time] instead of \$arr[\$time].

Range iterator outside integer range

(F) One (or both) of the numeric arguments to the range operator ".." are outside the range which can be represented by integers internally. One possible workaround is to force Perl to use magical string increment by prepending "0" to your numbers.

Recursive inheritance detected while looking for method '%s' %s

(F) More than 100 levels of inheritance were encountered while invoking a method. Probably indicates an unintended loop in your inheritance hierarchy.

Reference found where even-sized list expected

(W) You gave a single reference where Perl was expecting a list with an even number of elements (for assignment to a hash). This usually means that you used the anon hash constructor when you meant to use parens. In any case, a hash requires key/value **pairs**.

```
%hash = { one => 1, two => 2, };    # WRONG
%hash = [ qw/ an anon array / ];    # WRONG
%hash = ( one => 1, two => 2, );    # right
```

```
%hash = qw( one 1 two 2 );           # also fine
```

Undefined value assigned to typeglob

(W) An undefined value was assigned to a typeglob, a `la *foo = undef`. This does nothing. It's possible that you really mean `undef *foo`.

Use of reserved word "%s" is deprecated

(D) The indicated bareword is a reserved word. Future versions of perl may use it as a keyword, so you're better off either explicitly quoting the word in a manner appropriate for its context of use, or using a different name altogether. The warning can be suppressed for subroutine names by either adding a `&` prefix, or using a package qualifier, e.g. `&our()`, or `Foo::our()`.

perl: warning: Setting locale failed.

(S) The whole warning message will look something like:

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
    LC_ALL = "En_US",
    LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

Exactly what were the failed locale settings varies. In the above the settings were that the `LC_ALL` was "En_US" and the `LANG` had no value. This error means that Perl detected that you and/or your system administrator have set up the so-called variable system but Perl could not use those settings. This was not dead serious, fortunately: there is a "default locale" called "C" that Perl can and will use, the script will be run. Before you really fix the problem, however, you will get the same error message each time you run Perl. How to really fix the problem can be found in

[LOCALE PROBLEMS in perllocale](#).

Obsolete Diagnostics

Can't mktemp()

(F) The `mktemp()` routine failed for some reason while trying to process a `-e` switch. Maybe your `/tmp` partition is full, or clobbered.

Removed because `-e` doesn't use temporary files any more.

Can't write to temp file for `-e`: %s

(F) The write routine failed for some reason while trying to process a `-e` switch. Maybe your `/tmp` partition is full, or clobbered.

Removed because `-e` doesn't use temporary files any more.

Cannot open temporary file

(F) The create routine failed for some reason while trying to process a `-e` switch. Maybe your `/tmp` partition is full, or clobbered.

Removed because `-e` doesn't use temporary files any more.

regexp too big

(F) The current implementation of regular expressions uses shorts as address offsets within a string. Unfortunately this means that if the regular expression compiles to longer than 32767, it'll blow up. Usually when you want a regular expression this big, there is a better way to do it with multiple statements. See [perlre](#).

Configuration Changes

You can use "Configure `-Uinstallusrbinperl`" which causes `installperl` to skip installing perl also as `/usr/bin/perl`. This is useful if you prefer not to modify `/usr/bin` for some reason or another but harmful because many scripts assume to find Perl in `/usr/bin/perl`.

BUGS

If you find what you think is a bug, you might check the headers of recently posted articles in the comp.lang.perl.misc newsgroup. There may also be information at <http://www.perl.com/perl/> , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Make sure you trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to `<perlbug@perl.com>` to be analysed by the Perl porting team.

SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

HISTORY

Written by Gurusamy Sarathy `<gsar@activestate.com>`, with many contributions from The Perl Porters.

Send omissions or corrections to `<perlbug@perl.com>`.

NAME

perl561delta – what's new for perl v5.6.x

DESCRIPTION

This document describes differences between the 5.005 release and the 5.6.1 release.

Summary of changes between 5.6.0 and 5.6.1

This section contains a summary of the changes between the 5.6.0 release and the 5.6.1 release. More details about the changes mentioned here may be found in the *Changes* files that accompany the Perl source distribution. See *perlhack* for pointers to online resources where you can inspect the individual patches described by these changes.

Security Issues

suidperl will not run /bin/mail anymore, because some platforms have a /bin/mail that is vulnerable to buffer overflow attacks.

Note that suidperl is neither built nor installed by default in any recent version of perl. Use of suidperl is highly discouraged. If you think you need it, try alternatives such as sudo first. See <http://www.courtesan.com/sudo/>.

Core bug fixes

This is not an exhaustive list. It is intended to cover only the significant user-visible changes.

`UNIVERSAL::isa()`

A bug in the caching mechanism used by `UNIVERSAL::isa()` that affected `base.pm` has been fixed. The bug has existed since the 5.005 releases, but wasn't tickled by `base.pm` in those releases.

Memory leaks

Various cases of memory leaks and attempts to access uninitialized memory have been cured. See below for further issues.

Numeric conversions

Numeric conversions did not recognize changes in the string value properly in certain circumstances.

In other situations, large unsigned numbers (those above 2^{31}) could sometimes lose their unsignedness, causing bogus results in arithmetic operations.

Integer modulus on large unsigned integers sometimes returned incorrect values.

Perl 5.6.0 generated "not a number" warnings on certain conversions where previous versions didn't.

These problems have all been rectified.

Infinity is now recognized as a number.

`qw(a\\b)`

In Perl 5.6.0, `qw(a\\b)` produced a string with two backslashes instead of one, in a departure from the behavior in previous versions. The older behavior has been reinstated.

`caller()`

`caller()` could cause core dumps in certain situations. `Carp` was sometimes affected by this problem.

Bugs in regular expressions

Pattern matches on overloaded values are now handled correctly.

Perl 5.6.0 parsed `m/\x{ab}/` incorrectly, leading to spurious warnings. This has been corrected.

The RE engine found in Perl 5.6.0 accidentally pessimised certain kinds of simple pattern matches. These are now handled better.

Regular expression debug output (whether through `use re 'debug'` or via `-Dr`) now looks better.

Multi-line matches like `"a\nxb\n" =~ /(?!\\A)x/m` were flawed. The bug has been fixed.

Use of `$&` could trigger a core dump under some situations. This is now avoided.

Match variables `$1` et al., weren't being unset when a pattern match was backtracking, and the anomaly showed up inside `/... (?{ ... }) .../` etc. These variables are now tracked correctly.

`pos()` did not return the correct value within `s///ge` in earlier versions. This is now handled correctly.

"slurp" mode

`readline()` on files opened in "slurp" mode could return an extra `""` at the end in certain situations. This has been corrected.

Autovivification of symbolic references to special variables

Autovivification of symbolic references of special variables described in [perlvar](#) (as in `${ $num }`) was accidentally disabled. This works again now.

Lexical warnings

Lexical warnings now propagate correctly into `eval "..."`.

`use warnings qw(FATAL all)` did not work as intended. This has been corrected.

Lexical warnings could leak into other scopes in some situations. This is now fixed.

`warnings::enabled()` now reports the state of `$^W` correctly if the caller isn't using lexical warnings.

Spurious warnings and errors

Perl 5.6.0 could emit spurious warnings about redefinition of `dl_error()` when statically building extensions into perl. This has been corrected.

"our" variables could result in bogus "Variable will not stay shared" warnings. This is now fixed.

"our" variables of the same name declared in two sibling blocks resulted in bogus warnings about "redeclaration" of the variables. The problem has been corrected.

glob()

Compatibility of the builtin `glob()` with old csh-based `glob` has been improved with the addition of `GLOB_ALPHASORT` option. See `File::Glob`.

`File::Glob::glob()` has been renamed to `File::Glob::bsd_glob()` because the name clashes with the builtin `glob()`. The older name is still available for compatibility, but is deprecated.

Spurious syntax errors generated in certain situations, when `glob()` caused `File::Glob` to be loaded for the first time, have been fixed.

Tainting

Some cases of inconsistent taint propagation (such as within hash values) have been fixed.

The tainting behavior of `sprintf()` has been rationalized. It does not taint the result of floating point formats anymore, making the behavior consistent with that of string interpolation.

sort()

Arguments to `sort()` weren't being provided the right `wantarray()` context. The comparison block is now run in scalar context, and the arguments to be sorted are always provided list context.

`sort()` is also fully reentrant, in the sense that the sort function can itself call `sort()`. This did not work reliably in previous releases.

#line directives

#line directives now work correctly when they appear at the very beginning of `eval "..."`.

Subroutine prototypes

The `(&)` prototype now works properly.

map()

`map()` could get pathologically slow when the result list it generates is larger than the source list. The performance has been improved for common scenarios.

Debugger

Debugger exit code now reflects the script exit code.

Condition `"0"` in breakpoints is now treated correctly.

The `d` command now checks the line number.

`$.` is no longer corrupted by the debugger.

All debugger output now correctly goes to the socket if `RemotePort` is set.

PERL5OPT

`PERL5OPT` can be set to more than one switch group. Previously, it used to be limited to one group of options only.

chop()

`chop(@list)` in list context returned the characters chopped in reverse order. This has been reversed to be in the right order.

Unicode support

Unicode support has seen a large number of incremental improvements, but continues to be highly experimental. It is not expected to be fully supported in the 5.6.x maintenance releases.

`substr()`, `join()`, `repeat()`, `reverse()`, `quotemeta()` and string concatenation were all handling Unicode strings incorrectly in Perl 5.6.0. This has been corrected.

Support for `tr///CU` and `tr///UC` etc., have been removed since we realized the interface is broken. For similar functionality, see [pack](#).

The Unicode Character Database has been updated to version 3.0.1 with additions made available to the public as of August 30, 2000.

The Unicode character classes `\p{Blank}` and `\p{SpacePerl}` have been added. "Blank" is like `C isblank()`, that is, it contains only "horizontal whitespace" (the space character is, the newline isn't), and the "SpacePerl" is the Unicode equivalent of `\s` (`\p{Space}` isn't, since that includes the vertical tabulator character, whereas `\s` doesn't.)

If you are experimenting with Unicode support in perl, the development versions of Perl may have more to offer. In particular, I/O layers are now available in the development track, but not in the maintenance track, primarily to do backward compatibility issues. Unicode support is also evolving rapidly on a daily basis in the development track—the maintenance track only reflects the most conservative of these changes.

64-bit support

Support for 64-bit platforms has been improved, but continues to be experimental. The level of support varies greatly among platforms.

Compiler

The B Compiler and its various backends have had many incremental improvements, but they continue to remain highly experimental. Use in production environments is discouraged.

The perlcc tool has been rewritten so that the user interface is much more like that of a C compiler.

The perlbc tools has been removed. Use `perlcc -B` instead.

Lvalue subroutines

There have been various bugfixes to support lvalue subroutines better. However, the feature still remains experimental.

IO::Socket

IO::Socket::INET failed to open the specified port if the service name was not known. It now correctly uses the supplied port number as is.

File::Find

File::Find now `chdir()`s correctly when chasing symbolic links.

xsubpp

xsubpp now tolerates embedded POD sections.

no Module;

`no Module;` does not produce an error even if `Module` does not have an `unimport()` method. This parallels the behavior of `use` vis-a-vis `import`.

Tests

A large number of tests have been added.

Core features

`untie()` will now call an `UNTIE()` hook if it exists. See [perltie](#) for details.

The `-DT` command line switch outputs copious tokenizing information. See [perlrun](#).

Arrays are now always interpolated in double-quotish strings. Previously, `"foo@bar.com"` used to be a fatal error at compile time, if an array `@bar` was not used or declared. This transitional behavior was intended to help migrate perl4 code, and is deemed to be no longer useful. See .

`keys()`, `each()`, `pop()`, `push()`, `shift()`, `splice()` and `unshift()` can all be overridden now.

`my __PACKAGE__ $obj` now does the expected thing.

Configuration issues

On some systems (IRIX and Solaris among them) the system malloc is demonstrably better. While the defaults haven't been changed in order to retain binary compatibility with earlier releases, you may be better off building perl with `Configure -Uusemymalloc ...` as discussed in the **INSTALL** file.

Configure has been enhanced in various ways:

- Minimizes use of temporary files.
- By default, does not link perl with libraries not used by it, such as the various dbm libraries. SunOS 4.x hints preserve behavior on that platform.
- Support for pdp11-style memory models has been removed due to obsolescence.
- Building outside the source tree is supported on systems that have symbolic links. This is done by running

```
sh /path/to/source/Configure -Dmk symlinks ...
make all test install
```

in a directory other than the perl source directory. See **INSTALL**.

- `Configure -S` can be run non-interactively.

Documentation

README.aix, README.solaris and README.macos have been added. README.posix-bc has been renamed to README.bs2000. These are installed as [perlaix](#), [perlsolaris](#), [perlmacos](#), and [perlbs2000](#) respectively.

The following pod documents are brand new:

perlclib	Internal replacements for standard C library functions
perldebtut	Perl debugging tutorial
perlebcdic	Considerations for running Perl on EBCDIC platforms
perlnewmod	Perl modules: preparing a new module for distribution
perlrequick	Perl regular expressions quick start
perlretut	Perl regular expressions tutorial
perlutil	utilities packaged with the Perl distribution

The **INSTALL** file has been expanded to cover various issues, such as 64-bit support.

A longer list of contributors has been added to the source distribution. See the file AUTHORS.

Numerous other changes have been made to the included documentation and FAQs.

Bundled modules

The following modules have been added.

B::Concise

Walks Perl syntax tree, printing concise info about ops. See [B::Concise](#).

File::Temp

Returns name and handle of a temporary file safely. See [File::Temp](#).

Pod::LaTeX

Converts Pod data to formatted LaTeX. See [Pod::LaTeX](#).

Pod::Text::Overstrike

Converts POD data to formatted overstrike text. See [Pod::Text::Overstrike](#).

The following modules have been upgraded.

CGI CGI v2.752 is now included.

CPAN

CPAN v1.59_54 is now included.

Class::Struct

Various bugfixes have been added.

DB_File

DB_File v1.75 supports newer Berkeley DB versions, among other improvements.

Devel::Peek

Devel::Peek has been enhanced to support dumping of memory statistics, when perl is built with the included `malloc()`.

File::Find

File::Find now supports pre and post-processing of the files in order to `sort()` them, etc.

Getopt::Long

Getopt::Long v2.25 is included.

IO::Poll

Various bug fixes have been included.

IPC::Open3

IPC::Open3 allows use of numeric file descriptors.

Math::BigFloat

The `fmod()` function supports modulus operations. Various bug fixes have also been included.

Math::Complex

Math::Complex handles `inf`, `NaN` etc., better.

Net::Ping

`ping()` could fail on odd number of data bytes, and when the echo service isn't running. This has been corrected.

Opcode

A memory leak has been fixed.

Pod::Parser

Version 1.13 of the Pod::Parser suite is included.

Pod::Text

Pod::Text and related modules have been upgraded to the versions in podlators suite v2.08.

SDBM_File

On dosish platforms, some keys went missing because of lack of support for files with "holes". A workaround for the problem has been added.

Sys::Syslog

Various bug fixes have been included.

Tie::RefHash

Now supports Tie::RefHash::Nestable to automagically tie hashref values.

Tie::SubstrHash

Various bug fixes have been included.

Platform-specific improvements

The following new ports are now available.

NCR MP-RAS

NonStop-UX

Perl now builds under Amdahl UTS.

Perl has also been verified to build under Amiga OS.

Support for EPOC has been much improved. See README.epoc.

Building perl with `-Duseithreads` or `-Duse5005threads` now works under HP-UX 10.20 (previously it only worked under 10.30 or later). You will need a thread library package installed. See README.hpux.

Long doubles should now work under Linux.

Mac OS Classic is now supported in the mainstream source package. See README.macos.

Support for MPE/iX has been updated. See README.mpeix.

Support for OS/2 has been improved. See `os2/Changes` and README.os2.

Dynamic loading on z/OS (formerly OS/390) has been improved. See README.os390.

Support for VMS has seen many incremental improvements, including better support for operators like `backticks` and `system()`, and better `%ENV` handling. See `README.vms` and [perlvms](#).

Support for Stratus VOS has been improved. See `vos/Changes` and `README.vos`.

Support for Windows has been improved.

- `fork()` emulation has been improved in various ways, but still continues to be experimental. See [perlfork](#) for known bugs and caveats.
- `%SIG` has been enabled under `USE_ITHREADS`, but its use is completely unsupported under all configurations.
- Borland C++ v5.5 is now a supported compiler that can build Perl. However, the generated binaries continue to be incompatible with those generated by the other supported compilers (GCC and Visual C++).
- Non-blocking waits for child processes (or pseudo-processes) are supported via `waitpid($pid, &POSIX::WNOHANG)`.
- A memory leak in `accept()` has been fixed.
- `wait()`, `waitpid()` and `backticks` now return the correct exit status under Windows 9x.
- Trailing new `%ENV` entries weren't propagated to child processes. This is now fixed.
- Current directory entries in `%ENV` are now correctly propagated to child processes.
- Duping socket handles with `open(F, "&MYSOCK")` now works under Windows 9x.
- The makefiles now provide a single switch to bulk-enable all the features enabled in ActiveState ActivePerl (a popular binary distribution).
- `Win32::GetCwd()` correctly returns `C:\` instead of `C:` when at the drive root. Other bugs in `chdir()` and `Cwd::cwd()` have also been fixed.
- `fork()` correctly returns `undef` and sets `EAGAIN` when it runs out of pseudo-process handles.
- `ExtUtils::MakeMaker` now uses `$ENV{LIB}` to search for libraries.
- UNC path handling is better when perl is built to support `fork()`.
- A handle leak in socket handling has been fixed.
- `send()` works from within a pseudo-process.

Unless specifically qualified otherwise, the remainder of this document covers changes between the 5.005 and 5.6.0 releases.

Core Enhancements

Interpreter cloning, threads, and concurrency

Perl 5.6.0 introduces the beginnings of support for running multiple interpreters concurrently in different threads. In conjunction with the `perl_clone()` API call, which can be used to selectively duplicate the state of any given interpreter, it is possible to compile a piece of code once in an interpreter, clone that interpreter one or more times, and run all the resulting interpreters in distinct threads.

On the Windows platform, this feature is used to emulate `fork()` at the interpreter level. See [perlfork](#) for details about that.

This feature is still in evolution. It is eventually meant to be used to selectively clone a subroutine and data reachable from that subroutine in a separate interpreter and run the cloned subroutine in a separate thread. Since there is no shared data between the interpreters, little or no locking will be needed (unless parts of the symbol table are explicitly shared). This is obviously intended to be an easy-to-use replacement for the existing threads support.

Support for cloning interpreters and interpreter concurrency can be enabled using the `-Dusethreads` Configure option (see `win32/Makefile` for how to enable it on Windows.) The resulting perl executable will be functionally identical to one that was built with `-Dmultiplicity`, but the `perl_clone()` API call will only be available in the former.

`-Dusethreads` enables the cpp macro `USE_ITHREADS` by default, which in turn enables Perl source code changes that provide a clear separation between the op tree and the data it operates with. The former is immutable, and can therefore be shared between an interpreter and all of its clones, while the latter is considered local to each interpreter, and is therefore copied for each clone.

Note that building Perl with the `-Dusemultiplicity` Configure option is adequate if you wish to run multiple **independent** interpreters concurrently in different threads. `-Dusethreads` only provides the additional functionality of the `perl_clone()` API call and other support for running **cloned** interpreters concurrently.

NOTE: This is an experimental feature. Implementation details are subject to change.

Lexically scoped warning categories

You can now control the granularity of warnings emitted by perl at a finer level using the `use warnings` pragma. [warnings](#) and [perllexwarn](#) have copious documentation on this feature.

Unicode and UTF-8 support

Perl now uses UTF-8 as its internal representation for character strings. The `utf8` and `bytes` pragmas are used to control this support in the current lexical scope. See [perlunicode](#), [utf8](#) and [bytes](#) for more information.

This feature is expected to evolve quickly to support some form of I/O disciplines that can be used to specify the kind of input and output data (bytes or characters). Until that happens, additional modules from CPAN will be needed to complete the toolkit for dealing with Unicode.

NOTE: This should be considered an experimental feature. Implementation details are subject to change.

Support for interpolating named characters

The new `\N` escape interpolates named characters within strings. For example, `"Hi! \N{WHITE SMILING FACE}"` evaluates to a string with a Unicode smiley face at the end.

"our" declarations

An "our" declaration introduces a value that can be best understood as a lexically scoped symbolic alias to a global variable in the package that was current where the variable was declared. This is mostly useful as an alternative to the `vars` pragma, but also provides the opportunity to introduce typing and other attributes for such variables. See [our](#).

Support for strings represented as a vector of ordinals

Literals of the form `v1.2.3.4` are now parsed as a string composed of characters with the specified ordinals. This is an alternative, more readable way to construct (possibly Unicode) strings instead of interpolating characters, as in `"\x{1}\x{2}\x{3}\x{4}"`. The leading `v` may be omitted if there are more than two ordinals, so `1.2.3` is parsed the same as `v1.2.3`.

Strings written in this form are also useful to represent version "numbers". It is easy to compare such version "numbers" (which are really just plain strings) using any of the usual string comparison operators `eq`, `ne`, `lt`, `gt`, etc., or perform bitwise string operations on them using `|`, `&`, etc.

In conjunction with the new `$_V` magic variable (which contains the perl version as a string), such literals can be used as a readable way to check if you're running a particular version of Perl:

```
# this will parse in older versions of Perl also
if ($_V and $_V gt v5.6.0) {
    # new features supported
```

```
}
```

`require` and `use` also have some special magic to support such literals. They will be interpreted as a version rather than as a module name:

```
require v5.6.0;           # croak if $^V lt v5.6.0
use v5.6.0;               # same, but croaks at compile-time
```

Alternatively, the `v` may be omitted if there is more than one dot:

```
require 5.6.0;
use 5.6.0;
```

Also, `sprintf` and `printf` support the Perl-specific format flag `%v` to print ordinals of characters in arbitrary strings:

```
printf "v%vd", $^V;       # prints current version, such as "v5.5.650"
printf "%*vX", ":", $addr; # formats IPv6 address
printf "%*vb", " ", $bits; # displays bitstring
```

See [Scalar value constructors in perldata](#) for additional information.

Improved Perl version numbering system

Beginning with Perl version 5.6.0, the version number convention has been changed to a "dotted integer" scheme that is more commonly found in open source projects.

Maintenance versions of v5.6.0 will be released as v5.6.1, v5.6.2 etc. The next development series following v5.6.0 will be numbered v5.7.x, beginning with v5.7.0, and the next major production release following v5.6.0 will be v5.8.0.

The English module now sets `$PERL_VERSION` to `$^V` (a string value) rather than `$]` (a numeric value). (This is a potential incompatibility. Send us a report via perlbug if you are affected by this.)

The v1.2.3 syntax is also now legal in Perl. See [Support for strings represented as a vector of ordinals](#) for more on that.

To cope with the new versioning system's use of at least three significant digits for each version component, the method used for incrementing the subversion number has also changed slightly. We assume that versions older than v5.6.0 have been incrementing the subversion component in multiples of 10. Versions after v5.6.0 will increment them by 1. Thus, using the new notation, 5.005_03 is the "same" as v5.5.30, and the first maintenance version following v5.6.0 will be v5.6.1 (which should be read as being equivalent to a floating point value of 5.006_001 in the older format, stored in `$]`).

New syntax for declaring subroutine attributes

Formerly, if you wanted to mark a subroutine as being a method call or as requiring an automatic `lock()` when it is entered, you had to declare that with a `use attrs` pragma in the body of the subroutine. That can now be accomplished with declaration syntax, like this:

```
sub mymethod : locked method ;
...
sub mymethod : locked method {
    ...
}

sub othermethod :locked :method ;
...
sub othermethod :locked :method {
    ...
}
```

(Note how only the first `:` is mandatory, and whitespace surrounding the `:` is optional.)

AutoSplit.pm and *SelfLoader.pm* have been updated to keep the attributes with the stubs they provide. See [attributes](#).

File and directory handles can be autovivified

Similar to how constructs such as `< $x-[0]` autovivify a reference, handle constructors (`open()`, `opendir()`, `pipe()`, `socketpair()`, `sysopen()`, `socket()`, and `accept()`) now autovivify a file or directory handle if the handle passed to them is an uninitialized scalar variable. This allows the constructs such as `open(my $fh, ...)` and `open(local $fh, ...)` to be used to create filehandles that will conveniently be closed automatically when the scope ends, provided there are no other references to them. This largely eliminates the need for typeglobs when opening filehandles that must be passed around, as in the following example:

```
sub myopen {
    open my $fh, "@_"
        or die "Can't open '@_': $!";
    return $fh;
}

{
    my $f = myopen("</etc/motd");
    print <$f>;
    # $f implicitly closed here
}
```

`open()` with more than two arguments

If `open()` is passed three arguments instead of two, the second argument is used as the mode and the third argument is taken to be the file name. This is primarily useful for protecting against unintended magic behavior of the traditional two-argument form. See [open](#).

64-bit support

Any platform that has 64-bit integers either

- (1) natively as longs or ints
- (2) via special compiler flags
- (3) using long long or int64_t

is able to use "quads" (64-bit integers) as follows:

- constants (decimal, hexadecimal, octal, binary) in the code
- arguments to `oct()` and `hex()`
- arguments to `print()`, `printf()` and `sprintf()` (flag prefixes ll, L, q)
- printed as such
- `pack()` and `unpack()` "q" and "Q" formats
- in basic arithmetics: `+` `-` `*` `/` `%` (NOTE: operating close to the limits of the integer values may produce surprising results)
- in bit arithmetics: `&` `|` `^` `~` `<<` (NOTE: these used to be forced to be 32 bits wide but now operate on the full native width.)
- `vec()`

Note that unless you have the case (a) you will have to configure and compile Perl using the `-Duse64bitint` Configure flag.

NOTE: The Configure flags `-DuselONGLONG` and `-Duse64bits` have been deprecated. Use `-Duse64bitint` instead.

There are actually two modes of 64-bitness: the first one is achieved using `Configure -Duse64bitint` and the second one using `Configure -Duse64bitall`. The difference is that the first one is minimal and the second one maximal. The first works in more places than the second.

The `use64bitint` does only as much as is required to get 64-bit integers into Perl (this may mean, for example, using "long longs") while your memory may still be limited to 2 gigabytes (because your pointers could still be 32-bit). Note that the name `64bitint` does not imply that your C compiler will be using 64-bit ints (it might, but it doesn't have to): the `use64bitint` means that you will be able to have 64 bits wide scalar values.

The `use64bitall` goes all the way by attempting to switch also integers (if it can), longs (and pointers) to being 64-bit. This may create an even more binary incompatible Perl than `-Duse64bitint`: the resulting executable may not run at all in a 32-bit box, or you may have to reboot/reconfigure/rebuild your operating system to be 64-bit aware.

Natively 64-bit systems like Alpha and Cray need neither `-Duse64bitint` nor `-Duse64bitall`.

Last but not least: note that due to Perl's habit of always using floating point numbers, the quads are still not true integers. When quads overflow their limits (0...18_446_744_073_709_551_615 unsigned, -9_223_372_036_854_775_808...9_223_372_036_854_775_807 signed), they are silently promoted to floating point numbers, after which they will start losing precision (in their lower digits).

NOTE: 64-bit support is still experimental on most platforms. Existing support only covers the LP64 data model. In particular, the LLP64 data model is not yet supported. 64-bit libraries and system APIs on many platforms have not stabilized--your mileage may vary.

Large file support

If you have filesystems that support "large files" (files larger than 2 gigabytes), you may now also be able to create and access them from Perl.

NOTE: The default action is to enable large file support, if available on the platform.

If the large file support is on, and you have a `Fcntl` constant `O_LARGEFILE`, the `O_LARGEFILE` is automatically added to the flags of `sysopen()`.

Beware that unless your filesystem also supports "sparse files" seeking to umpteen petabytes may be inadvisable.

Note that in addition to requiring a proper file system to do large files you may also need to adjust your per-process (or your per-system, or per-process-group, or per-user-group) maximum filesize limits before running Perl scripts that try to handle large files, especially if you intend to write such files.

Finally, in addition to your process/process group maximum filesize limits, you may have quota limits on your filesystems that stop you (your user id or your user group id) from using large files.

Adjusting your process/user/group/file system/operating system limits is outside the scope of Perl core language. For process limits, you may try increasing the limits using your shell's `limits/limit/ulimit` command before running Perl. The BSD::Resource extension (not included with the standard Perl distribution) may also be of use, it offers the `getrlimit/setrlimit` interface that can be used to adjust process resource usage limits, including the maximum filesize limit.

Long doubles

In some systems you may be able to use long doubles to enhance the range and precision of your double precision floating point numbers (that is, Perl's numbers). Use `Configure -Duselongdouble` to enable this support (if it is available).

"more bits"

You can "`Configure -Dusemorebits`" to turn on both the 64-bit support and the long double support.

Enhanced support for `sort()` subroutines

Perl subroutines with a prototype of `($$)`, and XSUBs in general, can now be used as sort subroutines. In either case, the two elements to be compared are passed as normal parameters in `@_`. See [sort](#).

For unprototyped sort subroutines, the historical behavior of passing the elements to be compared as the global variables `$a` and `$b` remains unchanged.

`sort $coderef @foo` allowed

`sort()` did not accept a subroutine reference as the comparison function in earlier versions. This is now permitted.

File globbing implemented internally

Perl now uses the `File::Glob` implementation of the `glob()` operator automatically. This avoids using an external `csh` process and the problems associated with it.

NOTE: This is currently an experimental feature. Interfaces and implementation are subject to change.

Support for CHECK blocks

In addition to `BEGIN`, `INIT`, `END`, `DESTROY` and `AUTOLOAD`, subroutines named `CHECK` are now special. These are queued up during compilation and behave similar to `END` blocks, except they are called at the end of compilation rather than at the end of execution. They cannot be called directly.

POSIX character class syntax `[[:alpha:]]` supported

For example to match alphabetic characters use `/[[:alpha:]]/`. See [perlre](#) for details.

Better pseudo-random number generator

In 5.005_0x and earlier, perl's `rand()` function used the C library `rand(3)` function. As of 5.005_52, Configure tests for `drand48()`, `random()`, and `rand()` (in that order) and picks the first one it finds.

These changes should result in better random numbers from `rand()`.

Improved `qw//` operator

The `qw//` operator is now evaluated at compile time into a true list instead of being replaced with a run time call to `split()`. This removes the confusing misbehaviour of `qw//` in scalar context, which had inherited that behaviour from `split()`.

Thus:

```
$foo = ($bar) = qw(a b c); print "$foo|$bar\n";
```

now correctly prints "3|a", instead of "2|a".

Better worst-case behavior of hashes

Small changes in the hashing algorithm have been implemented in order to improve the distribution of lower order bits in the hashed value. This is expected to yield better performance on keys that are repeated sequences.

`pack()` format `'Z'` supported

The new format type `'Z'` is useful for packing and unpacking null-terminated strings. See [pack in perlfunc](#).

`pack()` format modifier `'!` supported

The new format type modifier `'!` is useful for packing and unpacking native shorts, ints, and longs. See [pack in perlfunc](#).

`pack()` and `unpack()` support counted strings

The template character `'/'` can be used to specify a counted string type to be packed or unpacked. See [pack in perlfunc](#).

Comments in `pack()` templates

The '#' character in a template introduces a comment up to end of the line. This facilitates documentation of `pack()` templates.

Weak references

In previous versions of Perl, you couldn't cache objects so as to allow them to be deleted if the last reference from outside the cache is deleted. The reference in the cache would hold a reference count on the object and the objects would never be destroyed.

Another familiar problem is with circular references. When an object references itself, its reference count would never go down to zero, and it would not get destroyed until the program is about to exit.

Weak references solve this by allowing you to "weaken" any reference, that is, make it not count towards the reference count. When the last non-weak reference to an object is deleted, the object is destroyed and all the weak references to the object are automatically undef-ed.

To use this feature, you need the `Devel::WeakRef` package from CPAN, which contains additional documentation.

NOTE: This is an experimental feature. Details are subject to change.

Binary numbers supported

Binary numbers are now supported as literals, in `s?printf` formats, and `oct()`:

```
$answer = 0b101010;  
printf "The answer is: %b\n", oct("0b101010");
```

Lvalue subroutines

Subroutines can now return modifiable lvalues. See [Lvalue subroutines in perlsyn](#).

NOTE: This is an experimental feature. Details are subject to change.

Some arrows may be omitted in calls through references

Perl now allows the arrow to be omitted in many constructs involving subroutine calls through references. For example, `< $foo[10] - ('foo') >` may now be written `$foo[10] ('foo')`. This is rather similar to how the arrow may be omitted from `< $foo[10] - { 'foo' } >`. Note however, that the arrow is still required for `< foo(10) - ('bar') >`.

Boolean assignment operators are legal lvalues

Constructs such as `($a ||= 2) += 1` are now allowed.

`exists()` is supported on subroutine names

The `exists()` builtin now works on subroutine names. A subroutine is considered to exist if it has been declared (even if implicitly). See [exists](#) for examples.

`exists()` and `delete()` are supported on array elements

The `exists()` and `delete()` builtins now work on simple arrays as well. The behavior is similar to that on hash elements.

`exists()` can be used to check whether an array element has been initialized. This avoids autovivifying array elements that don't exist. If the array is tied, the `EXISTS()` method in the corresponding tied package will be invoked.

`delete()` may be used to remove an element from the array and return it. The array element at that position returns to its uninitialized state, so that testing for the same element with `exists()` will return false. If the element happens to be the one at the end, the size of the array also shrinks up to the highest element that tests true for `exists()`, or 0 if none such is found. If the array is tied, the `DELETE()` method in the corresponding tied package will be invoked.

See [exists](#) and [delete](#) for examples.

Pseudo-hashes work better

Dereferencing some types of reference values in a pseudo-hash, such as `< $ph-{foo}[1]`, was accidentally disallowed. This has been corrected.

When applied to a pseudo-hash element, `exists()` now reports whether the specified value exists, not merely if the key is valid.

`delete()` now works on pseudo-hashes. When given a pseudo-hash element or slice it deletes the values corresponding to the keys (but not the keys themselves). See

[Pseudo-hashes: Using an array as a hash in perlref](#).

Pseudo-hash slices with constant keys are now optimized to array lookups at compile-time.

List assignments to pseudo-hash slices are now supported.

The `fields` pragma now provides ways to create pseudo-hashes, via `fields::new()` and `fields::phash()`. See *[fields](#)*.

NOTE: The pseudo-hash data type continues to be experimental. Limiting oneself to the interface elements provided by the `fields` pragma will provide protection from any future changes.

Automatic flushing of output buffers

`fork()`, `exec()`, `system()`, `qx//`, and pipe `open()`s now flush buffers of all files opened for output when the operation was attempted. This mostly eliminates confusing buffering mishaps suffered by users unaware of how Perl internally handles I/O.

This is not supported on some platforms like Solaris where a suitably correct implementation of `fflush(NULL)` isn't available.

Better diagnostics on meaningless filehandle operations

Constructs such as `< open(<FH)` and `< close(<FH)` are compile time errors. Attempting to read from filehandles that were opened only for writing will now produce warnings (just as writing to read-only filehandles does).

Where possible, buffered data discarded from duped input filehandle

`< open(NEW, "<&OLD")` now attempts to discard any data that was previously read and buffered in OLD before duping the handle. On platforms where doing this is allowed, the next read operation on NEW will return the same data as the corresponding operation on OLD. Formerly, it would have returned the data from the start of the following disk block instead.

`eof()` has the same old magic as `<`

`eof()` would return true if no attempt to read from `<` had yet been made. `eof()` has been changed to have a little magic of its own, it now opens the `<` files.

`binmode()` can be used to set `:crlf` and `:raw` modes

`binmode()` now accepts a second argument that specifies a discipline for the handle in question. The two pseudo-disciplines `":raw"` and `":crlf"` are currently supported on DOS-derivative platforms. See *[binmode in perlfunc](#)* and *[open](#)*.

`-T` filetest recognizes UTF-8 encoded files as "text"

The algorithm used for the `-T` filetest has been enhanced to correctly identify UTF-8 content as "text".

`system()`, backticks and pipe `open` now reflect `exec()` failure

On Unix and similar platforms, `system()`, `qx()` and `open(FOO, "cmd |")` etc., are implemented via `fork()` and `exec()`. When the underlying `exec()` fails, earlier versions did not report the error properly, since the `exec()` happened to be in a different process.

The child process now communicates with the parent about the error in launching the external command, which allows these constructs to return with their usual error value and set `$!`.

Improved diagnostics

Line numbers are no longer suppressed (under most likely circumstances) during the global destruction phase.

Diagnostics emitted from code running in threads other than the main thread are now accompanied by the thread ID.

Embedded null characters in diagnostics now actually show up. They used to truncate the message in prior versions.

`$foo:::a` and `$foo:::b` are now exempt from "possible typo" warnings only if `sort()` is encountered in package `foo`.

Unrecognized alphabetic escapes encountered when parsing quote constructs now generate a warning, since they may take on new semantics in later versions of Perl.

Many diagnostics now report the internal operation in which the warning was provoked, like so:

```
Use of uninitialized value in concatenation (.) at (eval 1) line 1.
Use of uninitialized value in print at (eval 1) line 1.
```

Diagnostics that occur within `eval` may also report the file and line number where the `eval` is located, in addition to the `eval` sequence number and the line number within the evaluated text itself. For example:

```
Not enough arguments for scalar at (eval 4) [newlib/perl5db.pl:1411] line 2, at EO
```

Diagnostics follow STDERR

Diagnostic output now goes to whichever file the `STDERR` handle is pointing at, instead of always going to the underlying C runtime library's `stderr`.

More consistent close-on-exec behavior

On systems that support a `close-on-exec` flag on filehandles, the flag is now set for any handles created by `pipe()`, `socketpair()`, `socket()`, and `accept()`, if that is warranted by the value of `$^F` that may be in effect. Earlier versions neglected to set the flag for handles created with these operators. See [pipe](#), [socketpair](#), [socket](#), [accept](#), and `$^F`.

`syswrite()` ease-of-use

The length argument of `syswrite()` has become optional.

Better syntax checks on parenthesized unary operators

Expressions such as:

```
print defined(&foo, &bar, &baz);
print uc("foo", "bar", "baz");
undef($foo, &bar);
```

used to be accidentally allowed in earlier versions, and produced unpredictable behaviour. Some produced ancillary warnings when used in this way; others silently did the wrong thing.

The parenthesized forms of most unary operators that expect a single argument now ensure that they are not called with more than one argument, making the cases shown above syntax errors. The usual behaviour of:

```
print defined &foo, &bar, &baz;
print uc "foo", "bar", "baz";
undef $foo, &bar;
```

remains unchanged. See [perlop](#).

Bit operators support full native integer width

The bit operators (`&`, `|`, `^`, `~`, `<<`) now operate on the full native integral width (the exact size of which is available in `$Config{ivsize}`). For example, if your platform is either natively 64-bit or if Perl has been configured to use 64-bit integers, these operations apply to 8 bytes (as opposed to 4 bytes on 32-bit

platforms). For portability, be sure to mask off the excess bits in the result of unary `~`, e.g., `~$x & 0xffffffff`.

Improved security features

More potentially unsafe operations taint their results for improved security.

The `passwd` and `shell` fields returned by the `getpwent()`, `getpwnam()`, and `getpwuid()` are now tainted, because the user can affect their own encrypted password and login shell.

The variable modified by `shmread()`, and messages returned by `msgrcv()` (and its object-oriented interface `IPC::SysV::Msg::rcv`) are also tainted, because other untrusted processes can modify messages and shared memory segments for their own nefarious purposes.

More functional bareword prototype (*)

Bareword prototypes have been rationalized to enable them to be used to override builtins that accept barewords and interpret them in a special way, such as `require` or `do`.

Arguments prototyped as `*` will now be visible within the subroutine as either a simple scalar or as a reference to a typeglob. See [Prototypes](#).

`require` and `do` may be overridden

`require` and `do` 'file' operations may be overridden locally by importing subroutines of the same name into the current package (or globally by importing them into the `CORE::GLOBAL::` namespace). Overriding `require` will also affect `use`, provided the override is visible at compile-time. See [Overriding Built-in Functions in perlsb](#).

`$^X` variables may now have names longer than one character

Formerly, `$^X` was synonymous with `${"\cX"}`, but `$^XY` was a syntax error. Now variable names that begin with a control character may be arbitrarily long. However, for compatibility reasons, these variables *must* be written with explicit braces, as `${^XY}` for example. `${^XYZ}` is synonymous with `${"\cXYZ"}`. Variable names with more than one control character, such as `${^XY^Z}`, are illegal.

The old syntax has not changed. As before, `^X` may be either a literal control-X character or the two-character sequence 'caret' plus 'X'. When braces are omitted, the variable name stops after the control character. Thus `"$^XYZ"` continues to be synonymous with `$^X . "YZ"` as before.

As before, lexical variables may not have names beginning with control characters. As before, variables whose names begin with a control character are always forced to be in package 'main'. All such variables are reserved for future extensions, except those that begin with `^_`, which may be used by user programs and are guaranteed not to acquire special meaning in any future version of Perl.

New variable `$^C` reflects `-c` switch

`$^C` has a boolean value that reflects whether perl is being run in compile-only mode (i.e. via the `-c` switch). Since `BEGIN` blocks are executed under such conditions, this variable enables perl code to determine whether actions that make sense only during normal running are warranted. See [perlvar](#).

New variable `$^V` contains Perl version as a string

`$^V` contains the Perl version number as a string composed of characters whose ordinals match the version numbers, i.e. v5.6.0. This may be used in string comparisons.

See `Support for strings represented as a vector of ordinals` for an example.

Optional Y2K warnings

If Perl is built with the cpp macro `PERL_Y2KWARN` defined, it emits optional warnings when concatenating the number 19 with another number.

This behavior must be specifically enabled when running Configure. See *INSTALL* and *README.Y2K*.

Arrays now always interpolate into double-quoted strings

In double-quoted strings, arrays now interpolate, no matter what. The behavior in earlier versions of perl 5 was that arrays would interpolate into strings if the array had been mentioned before the string was compiled, and otherwise Perl would raise a fatal compile-time error. In versions 5.000 through 5.003, the error was

```
Literal @example now requires backslash
```

In versions 5.004_01 through 5.6.0, the error was

```
In string, @example now must be written as \@example
```

The idea here was to get people into the habit of writing "fred\@example.com" when they wanted a literal @ sign, just as they have always written "Give me back my \\$5" when they wanted a literal \$ sign.

Starting with 5.6.1, when Perl now sees an @ sign in a double-quoted string, it *always* attempts to interpolate an array, regardless of whether or not the array has been used or declared already. The fatal error has been downgraded to an optional warning:

```
Possible unintended interpolation of @example in string
```

This warns you that "fred@example.com" is going to turn into fred.com if you don't backslash the @. See <http://www.plover.com/~mjd/perl/at-error.html> for more details about the history here.

Modules and Pragmata

Modules

attributes

While used internally by Perl as a pragma, this module also provides a way to fetch subroutine and variable attributes. See [attributes](#).

- B The Perl Compiler suite has been extensively reworked for this release. More of the standard Perl testsuite passes when run under the Compiler, but there is still a significant way to go to achieve production quality compiled executables.

NOTE: The Compiler suite remains highly experimental. The generated code may not be correct, even when it manages to execute without errors.

Benchmark

Overall, Benchmark results exhibit lower average error and better timing accuracy.

You can now run tests for *n* seconds instead of guessing the right number of tests to run: e.g., `timethese(-5, ...)` will run each code for at least 5 CPU seconds. Zero as the "number of repetitions" means "for at least 3 CPU seconds". The output format has also changed. For example:

```
use Benchmark;$x=3;timethese(-5,{a=>sub{$x*$x},b=>sub{$x**2}})
```

will now output something like this:

```
Benchmark: running a, b, each for at least 5 CPU seconds...
```

```
  a:  5 wallclock secs ( 5.77 usr +  0.00 sys =  5.77 CPU) @ 200551.91/
  b:  4 wallclock secs ( 5.00 usr +  0.02 sys =  5.02 CPU) @ 159605.18/
```

New features: "each for at least N CPU seconds...", "wallclock secs", and the "@ operations/CPU second (n=operations)".

`timethese()` now returns a reference to a hash of Benchmark objects containing the test results, keyed on the names of the tests.

`timethis()` now returns the iterations field in the Benchmark result object instead of 0.

`timethese()`, `timethis()`, and the new `cmpthese()` (see below) can also take a format specifier of 'none' to suppress output.

A new function `countit()` is just like `timeit()` except that it takes a `TIME` instead of a `COUNT`.

A new function `cmpthese()` prints a chart comparing the results of each test returned from a `timethese()` call. For each possible pair of tests, the percentage speed difference (iters/sec or seconds/iter) is shown.

For other details, see [Benchmark](#).

ByteLoader

The ByteLoader is a dedicated extension to generate and run Perl bytecode. See [ByteLoader](#).

constant

References can now be used.

The new version also allows a leading underscore in constant names, but disallows a double leading underscore (as in `"__LINE__"`). Some other names are disallowed or warned against, including `BEGIN`, `END`, etc. Some names which were forced into `main::` used to fail silently in some cases; now they're fatal (outside of `main::`) and an optional warning (inside of `main::`). The ability to detect whether a constant had been set with a given name has been added.

See [constant](#).

charnames

This pragma implements the `\N` string escape. See [charnames](#).

Data::Dumper

A `Maxdepth` setting can be specified to avoid venturing too deeply into deep data structures. See [Data::Dumper](#).

The XSUB implementation of `Dump()` is now automatically called if the `Useqq` setting is not in use.

Dumping `qr//` objects works correctly.

DB DB is an experimental module that exposes a clean abstraction to Perl's debugging API.

DB_File

DB_File can now be built with Berkeley DB versions 1, 2 or 3. See `ext/DB_File/Changes`.

Devel::DProf

Devel::DProf, a Perl source code profiler has been added. See [Devel::DProf](#) and [dprofpp](#).

Devel::Peek

The Devel::Peek module provides access to the internal representation of Perl variables and data. It is a data debugging tool for the XS programmer.

Dumpvalue

The Dumpvalue module provides screen dumps of Perl data.

DynaLoader

DynaLoader now supports a `dl_unload_file()` function on platforms that support unloading shared objects using `dlclose()`.

Perl can also optionally arrange to unload all extension shared objects loaded by Perl. To enable this, build Perl with the Configure option `-Accflags=-DDL_UNLOAD_ALL_AT_EXIT`. (This maybe useful if you are using Apache with `mod_perl`.)

English

`$PERL_VERSION` now stands for `$_V` (a string value) rather than for `$]` (a numeric value).

Env `Env` now supports accessing environment variables like `PATH` as array variables.

Fcntl

More `Fcntl` constants added: `F_SETLK64`, `F_SETLKW64`, `O_LARGEFILE` for large file (more than 4GB) access (NOTE: the `O_LARGEFILE` is automatically added to `sysopen()` flags if large file support has been configured, as is the default), Free/Net/OpenBSD locking behaviour flags `F_FLOCK`, `F_POSIX`, Linux `F_SHLCK`, and `O_ACCMODE`: the combined mask of `O_RDONLY`, `O_WRONLY`, and `O_RDWR`. The `seek()/sysseek()` constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` are available via the `:seek` tag. The `chmod()/stat()` `S_IF*` constants and `S_IS*` functions are available via the `:mode` tag.

File::Compare

A `compare_text()` function has been added, which allows custom comparison functions. See [File::Compare](#).

File::Find

`File::Find` now works correctly when the `wanted()` function is either autoloaded or is a symbolic reference.

A bug that caused `File::Find` to lose track of the working directory when pruning top-level directories has been fixed.

`File::Find` now also supports several other options to control its behavior. It can follow symbolic links if the `follow` option is specified. Enabling the `no_chdir` option will make `File::Find` skip changing the current directory when walking directories. The `untaint` flag can be useful when running with taint checks enabled.

See [File::Find](#).

File::Glob

This extension implements BSD-style file globbing. By default, it will also be used for the internal implementation of the `glob()` operator. See [File::Glob](#).

File::Spec

New methods have been added to the `File::Spec` module: `devnull()` returns the name of the null device (`/dev/null` on Unix) and `tmpdir()` the name of the temp directory (normally `/tmp` on Unix). There are now also methods to convert between absolute and relative filenames: `abs2rel()` and `rel2abs()`. For compatibility with operating systems that specify volume names in file paths, the `splitpath()`, `splitdir()`, and `catdir()` methods have been added.

File::Spec::Functions

The new `File::Spec::Functions` module provides a function interface to the `File::Spec` module. Allows shorthand

```
$fullname = catfile($dir1, $dir2, $file);
```

instead of

```
$fullname = File::Spec->catfile($dir1, $dir2, $file);
```

Getopt::Long

`Getopt::Long` licensing has changed to allow the Perl Artistic License as well as the GPL. It used to be GPL only, which got in the way of non-GPL applications that wanted to use `Getopt::Long`.

`Getopt::Long` encourages the use of `Pod::Usage` to produce help messages. For example:

```
use Getopt::Long;
```



```

use Pod::Usage;
my $man = 0;
my $help = 0;
GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
pod2usage(1) if $help;
pod2usage(-exitstatus => 0, -verbose => 2) if $man;

__END__

=head1 NAME

sample - Using Getopt::Long and Pod::Usage

=head1 SYNOPSIS

sample [options] [file ...]

Options:
    -help             brief help message
    -man              full documentation

=head1 OPTIONS

=over 8

=item B<-help>

Print a brief help message and exits.

=item B<-man>

Prints the manual page and exits.

=back

=head1 DESCRIPTION

B<This program> will read the given input file(s) and do something
useful with the contents thereof.

=cut

```

See [Pod::Usage](#) for details.

A bug that prevented the non-option call-back < from being specified as the first argument has been fixed.

To specify the characters < and as option starters, use <. Note, however, that changing option starters is strongly deprecated.

- IO write() and syswrite() will now accept a single-argument form of the call, for consistency with Perl's syswrite().

You can now create a TCP-based IO::Socket::INET without forcing a connect attempt. This allows you to configure its options (like making it non-blocking) and then call connect() manually.

A bug that prevented the IO::Socket::protocol() accessor from ever returning the correct value has been corrected.

IO::Socket::connect now uses non-blocking IO instead of alarm() to do connect timeouts.

IO::Socket::accept now uses select() instead of alarm() for doing timeouts.

IO::Socket::INET-new now sets \$! correctly on failure. \$@ is still set for backwards compatibility.

JPL Java Perl Lingo is now distributed with Perl. See `jpl/README` for more information.

lib `use lib` now weeds out any trailing duplicate entries. `no lib` removes all named entries.

Math::BigInt

The bitwise operations `<<`, `<<<`, `&`, `|`, and `~` are now supported on bigints.

Math::Complex

The accessor methods `Re`, `Im`, `arg`, `abs`, `rho`, and `theta` can now also act as mutators (accessor `$z-Re()`, mutator `$z-Re(3)`).

The class method `display_format` and the corresponding object method `display_format`, in addition to accepting just one argument, now can also accept a parameter hash. Recognized keys of a parameter hash are `"style"`, which corresponds to the old one parameter case, and two new parameters: `"format"`, which is a `printf()`-style format string (defaults usually to `"%.15g"`, you can revert to the default by setting the format string to `undef`) used for both parts of a complex number, and `"polar_pretty_print"` (defaults to `true`), which controls whether an attempt is made to try to recognize small multiples and rationals of π (2π , $\pi/2$) at the argument (angle) of a polar complex number.

The potentially disruptive change is that in list context both methods now *return the parameter hash*, instead of only the value of the `"style"` parameter.

Math::Trig

A little bit of radial trigonometry (cylindrical and spherical), radial coordinate conversions, and the great circle distance were added.

Pod::Parser, Pod::InputObjects

`Pod::Parser` is a base class for parsing and selecting sections of pod documentation from an input stream. This module takes care of identifying pod paragraphs and commands in the input and hands off the parsed paragraphs and commands to user-defined methods which are free to interpret or translate them as they see fit.

`Pod::InputObjects` defines some input objects needed by `Pod::Parser`, and for advanced users of `Pod::Parser` that need more about a command besides its name and text.

As of release 5.6.0 of Perl, `Pod::Parser` is now the officially sanctioned "base parser code" recommended for use by all `pod2xxx` translators. `Pod::Text` (`pod2text`) and `Pod::Man` (`pod2man`) have already been converted to use `Pod::Parser` and efforts to convert `Pod::HTML` (`pod2html`) are already underway. For any questions or comments about pod parsing and translating issues and utilities, please use the `pod-people@perl.org` mailing list.

For further information, please see [Pod::Parser](#) and [Pod::InputObjects](#).

Pod::Checker, podchecker

This utility checks pod files for correct syntax, according to [perlpod](#). Obvious errors are flagged as such, while warnings are printed for mistakes that can be handled gracefully. The checklist is not complete yet. See [Pod::Checker](#).

Pod::ParseUtils, Pod::Find

These modules provide a set of gizmos that are useful mainly for pod translators.

[Pod::Find](#)/[Pod::Find](#) traverses directory structures and returns found pod files, along with their canonical names (like `File::Spec::Unix`). [Pod::ParseUtils](#)/[Pod::ParseUtils](#) contains **Pod::List** (useful for storing pod list information), **Pod::Hyperlink** (for parsing the contents of `L<>` sequences) and **Pod::Cache** (for caching information about pod files, e.g., link nodes).

Pod::Select, podselect

`Pod::Select` is a subclass of `Pod::Parser` which provides a function named `"podselect ()"` to filter out user-specified sections of raw pod documentation from an input stream. `podselect` is a script that

provides access to `Pod::Select` from other scripts to be used as a filter. See [Pod::Select](#).

Pod::Usage, pod2usage

`Pod::Usage` provides the function `"pod2usage ()"` to print usage messages for a Perl script based on its embedded pod documentation. The `pod2usage ()` function is generally useful to all script authors since it lets them write and maintain a single source (the pods) for documentation, thus removing the need to create and maintain redundant usage message text consisting of information already in the pods.

There is also a `pod2usage` script which can be used from other kinds of scripts to print usage messages from pods (even for non-Perl scripts with pods embedded in comments).

For details and examples, please see [Pod::Usage](#).

Pod::Text and Pod::Man

`Pod::Text` has been rewritten to use `Pod::Parser`. While `pod2text ()` is still available for backwards compatibility, the module now has a new preferred interface. See [Pod::Text](#) for the details. The new `Pod::Text` module is easily subclassed for tweaks to the output, and two such subclasses (`Pod::Text::Termcap` for man-page-style bold and underlining using termcap information, and `Pod::Text::Color` for markup with ANSI color sequences) are now standard.

`pod2man` has been turned into a module, `Pod::Man`, which also uses `Pod::Parser`. In the process, several outstanding bugs related to quotes in section headers, quoting of code escapes, and nested lists have been fixed. `pod2man` is now a wrapper script around this module.

SDBM_File

An `EXISTS` method has been added to this module (and `sdbm_exists ()` has been added to the underlying `sdbm` library), so one can now call `exists` on an `SDBM_File` tied hash and get the correct result, rather than a runtime error.

A bug that may have caused data loss when more than one disk block happens to be read from the database in a single `FETCH ()` has been fixed.

Sys::Syslog

`Sys::Syslog` now uses XSUBs to access facilities from `syslog.h` so it no longer requires `syslog.ph` to exist.

Sys::Hostname

`Sys::Hostname` now uses XSUBs to call the C library's `gethostname ()` or `uname ()` if they exist.

Term::ANSIColor

`Term::ANSIColor` is a very simple module to provide easy and readable access to the ANSI color and highlighting escape sequences, supported by most ANSI terminal emulators. It is now included standard.

Time::Local

The `timelocal ()` and `timegm ()` functions used to silently return bogus results when the date fell outside the machine's integer range. They now consistently `croak ()` if the date falls in an unsupported range.

Win32

The error return value in list context has been changed for all functions that return a list of values. Previously these functions returned a list with a single element `undef` if an error occurred. Now these functions return the empty list in these situations. This applies to the following functions:

```
Win32::FsType
Win32::GetOSVersion
```

The remaining functions are unchanged and continue to return `undef` on error even in list context.

The `Win32::SetLastError(ERROR)` function has been added as a complement to the `Win32::GetLastError()` function.

The new `Win32::GetFullPathName(FILENAME)` returns the full absolute pathname for `FILENAME` in scalar context. In list context it returns a two-element list containing the fully qualified directory name and the filename. See [Win32](#).

XSLoader

The XSLoader extension is a simpler alternative to DynaLoader. See [XSLoader](#).

DBM Filters

A new feature called "DBM Filters" has been added to all the DBM modules—`DB_File`, `GDBM_File`, `NDBM_File`, `ODBM_File`, and `SDBM_File`. DBM Filters add four new methods to each DBM module:

```
filter_store_key
filter_store_value
filter_fetch_key
filter_fetch_value
```

These can be used to filter key-value pairs before the pairs are written to the database or just after they are read from the database. See [perldbfilter](#) for further information.

Pragmata

`use attrs` is now obsolete, and is only provided for backward-compatibility. It's been replaced by the `sub : attributes` syntax. See [Subroutine Attributes in perlsub](#) and [attributes](#).

Lexical warnings pragma, `use warnings;`, to control optional warnings. See [perllexwarn](#).

`use filetest` to control the behaviour of filetests (`-r -w ...`). Currently only one subpragma implemented, "use filetest 'access';", that uses `access(2)` or equivalent to check permissions instead of using `stat(2)` as usual. This matters in filesystems where there are ACLs (access control lists): the `stat(2)` might lie, but `access(2)` knows better.

The `open` pragma can be used to specify default disciplines for handle constructors (e.g. `open()`) and for `qx//`. The two pseudo-disciplines `:raw` and `:crlf` are currently supported on DOS-derivative platforms (i.e. where `binmode` is not a no-op). See also ["/binmode\(\)](#) can be used to set `:crlf` and `:raw` modes".

Utility Changes

dprofpp

`dprofpp` is used to display profile data generated using `Devel::DProf`. See [dprofpp](#).

find2perl

The `find2perl` utility now uses the enhanced features of the `File::Find` module. The `-depth` and `-follow` options are supported. Pod documentation is also included in the script.

h2xs

The `h2xs` tool can now work in conjunction with `C::Scan` (available from CPAN) to automatically parse real-life header files. The `-M`, `-a`, `-k`, and `-o` options are new.

perlcc

`perlcc` now supports the C and Bytecode backends. By default, it generates output from the simple C backend rather than the optimized C backend.

Support for non-Unix platforms has been improved.

perldoc

`perldoc` has been reworked to avoid possible security holes. It will not by default let itself be run as the superuser, but you may still use the `-U` switch to try to make it drop privileges first.

The Perl Debugger

Many bug fixes and enhancements were added to *perl5db.pl*, the Perl debugger. The help documentation was rearranged. New commands include `< < ?`, `< < ?`, and `< { ?` to list out current actions, `man docpage` to run your doc viewer on some perl docset, and support for quoted options. The help information was rearranged, and should be viewable once again if you're using **less** as your pager. A serious security hole was plugged—you should immediately remove all older versions of the Perl debugger as installed in previous releases, all the way back to perl3, from your system to avoid being bitten by this.

Improved Documentation

Many of the platform-specific README files are now part of the perl installation. See [perl](#) for the complete list.

perlapi.pod

The official list of public Perl API functions.

perlboot.pod

A tutorial for beginners on object-oriented Perl.

perlcompile.pod

An introduction to using the Perl Compiler suite.

perldbfilter.pod

A howto document on using the DBM filter facility.

perldebug.pod

All material unrelated to running the Perl debugger, plus all low-level guts-like details that risked crushing the casual user of the debugger, have been relocated from the old manpage to the next entry below.

perldebguts.pod

This new manpage contains excessively low-level material not related to the Perl debugger, but slightly related to debugging Perl itself. It also contains some arcane internal details of how the debugging process works that may only be of interest to developers of Perl debuggers.

perlfork.pod

Notes on the `fork()` emulation currently available for the Windows platform.

perlfilter.pod

An introduction to writing Perl source filters.

perlhack.pod

Some guidelines for hacking the Perl source code.

perlintern.pod

A list of internal functions in the Perl source code. (List is currently empty.)

perllexwarn.pod

Introduction and reference information about lexically scoped warning categories.

perlnumber.pod

Detailed information about numbers as they are represented in Perl.

perlopentut.pod

A tutorial on using `open()` effectively.

perlreftut.pod

A tutorial that introduces the essentials of references.

perltootc.pod

A tutorial on managing class data for object modules.

perltodo.pod

Discussion of the most often wanted features that may someday be supported in Perl.

perlunicode.pod

An introduction to Unicode support features in Perl.

Performance enhancements

Simple `sort()` using `{ $a <= $b }` and the like are optimized

Many common `sort()` operations using a simple inlined block are now optimized for faster performance.

Optimized assignments to lexical variables

Certain operations in the RHS of assignment statements have been optimized to directly set the lexical variable on the LHS, eliminating redundant copying overheads.

Faster subroutine calls

Minor changes in how subroutine calls are handled internally provide marginal improvements in performance.

`delete()`, `each()`, `values()` and hash iteration are faster

The hash values returned by `delete()`, `each()`, `values()` and hashes in a list context are the actual values in the hash, instead of copies. This results in significantly better performance, because it eliminates needless copying in most situations.

Installation and Configuration Improvements

–Dusethreads means something different

The `–Dusethreads` flag now enables the experimental interpreter-based thread support by default. To get the flavor of experimental threads that was in 5.005 instead, you need to run `Configure` with `"–Dusethreads –Duse5005threads"`.

As of v5.6.0, interpreter-threads support is still lacking a way to create new threads from Perl (i.e., `use Thread`; will not work with interpreter threads). `use Thread`; continues to be available when you specify the `–Duse5005threads` option to `Configure`, bugs and all.

NOTE: Support for threads continues to be an experimental feature.

Interfaces and implementation are subject to sudden and drastic changes.

New Configure flags

The following new flags may be enabled on the `Configure` command line by running `Configure` with `–Dflag`.

<code>usemultiplicity</code>	
<code>usethreads useithreads</code>	(new interpreter threads: no Perl API yet)
<code>usethreads use5005threads</code>	(threads as they were in 5.005)
<code>use64bitint</code>	(equal to now deprecated <code>'use64bits'</code>)
<code>use64bitall</code>	
<code>uselongdouble</code>	
<code>usemorebits</code>	
<code>uselargefiles</code>	
<code>usesocks</code>	(only SOCKS v5 supported)

Threadedness and 64-bitness now more daring

The `Configure` options enabling the use of threads and the use of 64-bitness are now more daring in the sense that they no more have an explicit list of operating systems of known threads/64-bit capabilities. In

other words: if your operating system has the necessary APIs and datatypes, you should be able just to go ahead and use them, for threads by `Configure -Dusethreads`, and for 64 bits either explicitly by `Configure -Duse64bitint` or implicitly if your system has 64-bit wide datatypes. See also *"64-bit support"*.

Long Doubles

Some platforms have "long doubles", floating point numbers of even larger range than ordinary "doubles". To enable using long doubles for Perl's scalars, use `-Duselongdouble`.

`-Dusemorebits`

You can enable both `-Duse64bitint` and `-Duselongdouble` with `-Dusemorebits`. See also *"64-bit support"*.

`-Duselargefiles`

Some platforms support system APIs that are capable of handling large files (typically, files larger than two gigabytes). Perl will try to use these APIs if you ask for `-Duselargefiles`.

See *"Large file support"* for more information.

`installusrbinperl`

You can use `"Configure -Uinstallusrbinperl"` which causes `installperl` to skip installing perl also as `/usr/bin/perl`. This is useful if you prefer not to modify `/usr/bin` for some reason or another but harmful because many scripts assume to find Perl in `/usr/bin/perl`.

SOCKS support

You can use `"Configure -Dusesocks"` which causes Perl to probe for the SOCKS proxy protocol library (v5, not v4). For more information on SOCKS, see:

<http://www.socks.nec.com/>

`-A` flag

You can "post-edit" the `Configure` variables using the `Configure -A` switch. The editing happens immediately after the platform specific hints files have been processed but before the actual configuration process starts. Run `Configure -h` to find out the full `-A` syntax.

Enhanced Installation Directories

The installation structure has been enriched to improve the support for maintaining multiple versions of perl, to provide locations for vendor-supplied modules, scripts, and manpages, and to ease maintenance of locally-added modules, scripts, and manpages. See the section on Installation Directories in the `INSTALL` file for complete details. For most users building and installing from source, the defaults should be fine.

If you previously used `Configure -Dsitelib` or `-Dsitearch` to set special values for library directories, you might wish to consider using the new `-Dsiteprefix` setting instead. Also, if you wish to re-use a `config.sh` file from an earlier version of perl, you should be sure to check that `Configure` makes sensible choices for the new directories. See `INSTALL` for complete details.

gcc automatically tried if 'cc' does not seem to be working

In many platforms the vendor-supplied 'cc' is too stripped-down to build Perl (basically, the 'cc' doesn't do ANSI C). If this seems to be the case and the 'cc' does not seem to be the GNU C compiler 'gcc', an automatic attempt is made to find and use 'gcc' instead.

Platform specific changes

Supported platforms

- The Mach CThreads (NEXTSTEP, OPENSTEP) are now supported by the Thread extension.
- GNU/Hurd is now supported.
- Rhapsody/Darwin is now supported.

- EPOC is now supported (on Psion 5).
- The cygwin port (formerly cygwin32) has been greatly improved.

DOS

- Perl now works with djgpp 2.02 (and 2.03 alpha).
- Environment variable names are not converted to uppercase any more.
- Incorrect exit codes from backticks have been fixed.
- This port continues to use its own builtin globbing (not File::Glob).

OS390 (OpenEdition MVS)

Support for this EBCDIC platform has not been renewed in this release. There are difficulties in reconciling Perl's standardization on UTF-8 as its internal representation for characters with the EBCDIC character set, because the two are incompatible.

It is unclear whether future versions will renew support for this platform, but the possibility exists.

VMS

Numerous revisions and extensions to configuration, build, testing, and installation process to accommodate core changes and VMS-specific options.

Expand %ENV-handling code to allow runtime mapping to logical names, CLI symbols, and CRTL environ array.

Extension of subprocess invocation code to accept filespecs as command "verbs".

Add to Perl command line processing the ability to use default file types and to recognize Unix-style 2>&1.

Expansion of File::Spec::VMS routines, and integration into ExtUtils::MM_VMS.

Extension of ExtUtils::MM_VMS to handle complex extensions more flexibly.

Barewords at start of Unix-syntax paths may be treated as text rather than only as logical names.

Optional secure translation of several logical names used internally by Perl.

Miscellaneous bugfixing and porting of new core code to VMS.

Thanks are gladly extended to the many people who have contributed VMS patches, testing, and ideas.

Win32

Perl can now emulate `fork()` internally, using multiple interpreters running in different concurrent threads. This support must be enabled at build time. See [perlfork](#) for detailed information.

When given a pathname that consists only of a drivename, such as `A:`, `opendir()` and `stat()` now use the current working directory for the drive rather than the drive root.

The builtin XSUB functions in the Win32:: namespace are documented. See [Win32](#).

`$^X` now contains the full path name of the running executable.

A `Win32::GetLongPathName()` function is provided to complement `Win32::GetFullPathName()` and `Win32::GetShortPathName()`. See [Win32](#).

`POSIX::uname()` is supported.

`system(1,...)` now returns true process IDs rather than process handles. `kill()` accepts any real process id, rather than strictly return values from `system(1,...)`.

For better compatibility with Unix, `kill(0, $pid)` can now be used to test whether a process exists.

The `Shell` module is supported.

Better support for building Perl under `command.com` in Windows 95 has been added.

Scripts are read in binary mode by default to allow `ByteLoader` (and the filter mechanism in general) to work properly. For compatibility, the `DATA` filehandle will be set to text mode if a carriage return is detected at the end of the line containing the `__END__` or `__DATA__` token; if not, the `DATA` filehandle will be left open in binary mode. Earlier versions always opened the `DATA` filehandle in text mode.

The `glob()` operator is implemented via the `File::Glob` extension, which supports `glob` syntax of the C shell. This increases the flexibility of the `glob()` operator, but there may be compatibility issues for programs that relied on the older globbing syntax. If you want to preserve compatibility with the older syntax, you might want to run perl with `-MFile::DosGlob`. For details and compatibility information, see [File::Glob](#).

Significant bug fixes

<HANDLE on empty files

With `$/` set to `undef`, "slurping" an empty file returns a string of zero length (instead of `undef`, as it used to) the first time the `HANDLE` is read after `$/` is set to `undef`. Further reads yield `undef`.

This means that the following will append "foo" to an empty file (it used to do nothing):

```
perl -0777 -pi -e 's/^/foo/' empty_file
```

The behaviour of:

```
perl -pi -e 's/^/foo/' empty_file
```

is unchanged (it continues to leave the file empty).

eval '...' improvements

Line numbers (as reflected by `caller()` and most diagnostics) within `eval '...'` were often incorrect where here documents were involved. This has been corrected.

Lexical lookups for variables appearing in `eval '...'` within functions that were themselves called within an `eval '...'` were searching the wrong place for lexicals. The lexical search now correctly ends at the subroutine's block boundary.

The use of `return` within `eval { ... }` caused `$@` not to be reset correctly when no exception occurred within the eval. This has been fixed.

Parsing of here documents used to be flawed when they appeared as the replacement expression in `eval 's/.../.../e'`. This has been fixed.

All compilation errors are true errors

Some "errors" encountered at compile time were by necessity generated as warnings followed by eventual termination of the program. This enabled more such errors to be reported in a single run, rather than causing a hard stop at the first error that was encountered.

The mechanism for reporting such errors has been reimplemented to queue compile-time errors and report them at the end of the compilation as true errors rather than as warnings. This fixes cases where error messages leaked through in the form of warnings when code was compiled at run time using `eval STRING`, and also allows such errors to be reliably trapped using `eval "..."`.

Implicitly closed filehandles are safer

Sometimes implicitly closed filehandles (as when they are localized, and Perl automatically closes them on exiting the scope) could inadvertently set `$?` or `$!`. This has been corrected.

Behavior of list slices is more consistent

When taking a slice of a literal list (as opposed to a slice of an array or hash), Perl used to return an empty list if the result happened to be composed of all `undef` values.

The new behavior is to produce an empty list if (and only if) the original list was empty. Consider the following example:

```
@a = (1, undef, undef, 2) [2, 1, 2];
```

The old behavior would have resulted in @a having no elements. The new behavior ensures it has three undefined elements.

Note in particular that the behavior of slices of the following cases remains unchanged:

```
@a = () [1, 2];
@a = (getpwent) [7, 0];
@a = (anything_returning_empty_list()) [2, 1, 2];
@a = @b [2, 1, 2];
@a = @c { 'a', 'b', 'c' };
```

See [perldata](#).

(\ \$) prototype and \$foo{a}

A scalar reference prototype now correctly allows a hash or array element in that slot.

goto &sub and AUTOLOAD

The goto &sub construct works correctly when &sub happens to be autoloading.

-bareword allowed under use integer

The autoquoting of barewords preceded by – did not work in prior versions when the integer pragma was enabled. This has been fixed.

Failures in DESTROY()

When code in a destructor threw an exception, it went unnoticed in earlier versions of Perl, unless someone happened to be looking in \$@ just after the point the destructor happened to run. Such failures are now visible as warnings when warnings are enabled.

Locale bugs fixed

printf() and sprintf() previously reset the numeric locale back to the default "C" locale. This has been fixed.

Numbers formatted according to the local numeric locale (such as using a decimal comma instead of a decimal dot) caused "isn't numeric" warnings, even while the operations accessing those numbers produced correct results. These warnings have been discontinued.

Memory leaks

The eval 'return sub { ... }' construct could sometimes leak memory. This has been fixed.

Operations that aren't filehandle constructors used to leak memory when used on invalid filehandles. This has been fixed.

Constructs that modified @_ could fail to deallocate values in @_ and thus leak memory. This has been corrected.

Spurious subroutine stubs after failed subroutine calls

Perl could sometimes create empty subroutine stubs when a subroutine was not found in the package. Such cases stopped later method lookups from progressing into base packages. This has been corrected.

Taint failures under -U

When running in unsafe mode, taint violations could sometimes cause silent failures. This has been fixed.

END blocks and the -c switch

Prior versions used to run BEGIN and END blocks when Perl was run in compile-only mode. Since this is typically not the expected behavior, END blocks are not executed anymore when the -c switch is used, or if compilation fails.

See `for` how to run things when the `compile` phase ends.

Potential to leak DATA filehandles

Using the `__DATA__` token creates an implicit filehandle to the file that contains the token. It is the program's responsibility to close it when it is done reading from it.

This caveat is now better explained in the documentation. See [perldata](#).

New or Changed Diagnostics

"%s" variable %s masks earlier declaration in same %s

(W misc) A "my" or "our" variable has been redeclared in the current scope or statement, effectively eliminating all access to the previous instance. This is almost always a typographical error. Note that the earlier variable will still exist until the end of the scope or until all closure referents to it are destroyed.

"my sub" not yet implemented

(F) Lexically scoped subroutines are not yet implemented. Don't try that yet.

"our" variable %s redeclared

(W misc) You seem to have already declared the same global once before in the current lexical scope.

'!' allowed only after types %s

(F) The '!' is allowed in `pack()` and `unpack()` only after certain types. See [pack](#).

/ cannot take a count

(F) You had an `unpack` template indicating a counted-length string, but you have also specified an explicit size for the string. See [pack](#).

/ must be followed by a, A or Z

(F) You had an `unpack` template indicating a counted-length string, which must be followed by one of the letters a, A or Z to indicate what sort of string is to be unpacked. See [pack](#).

/ must be followed by a*, A* or Z*

(F) You had a `pack` template indicating a counted-length string. Currently the only things that can have their length counted are a*, A* or Z*. See [pack](#).

/ must follow a numeric type

(F) You had an `unpack` template that contained a '#', but this did not follow some numeric `unpack` specification. See [pack](#).

/%s/: Unrecognized escape \\%c passed through

(W regexp) You used a backslash-character combination which is not recognized by Perl. This combination appears in an interpolated variable or a '-delimited regular expression. The character was understood literally.

/%s/: Unrecognized escape \\%c in character class passed through

(W regexp) You used a backslash-character combination which is not recognized by Perl inside character classes. The character was understood literally.

/%s/ should probably be written as "%s"

(W syntax) You have used a pattern where Perl expected to find a string, as in the first argument to `join`. Perl will treat the true or false result of matching the pattern against `$_` as the string, which is probably not what you had in mind.

%s() called too early to check prototype

(W prototype) You've called a function that has a prototype before the parser saw a definition or declaration for it, and Perl could not check that the call conforms to the prototype. You need to either add an early prototype declaration for the subroutine in question, or move the subroutine definition

ahead of the call to get proper prototype checking. Alternatively, if you are certain that you're calling the function correctly, you may put an ampersand before the name to avoid the warning. See [perlsyn](#).

%s argument is not a HASH or ARRAY element

(F) The argument to `exists()` must be a hash or array element, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
```

%s argument is not a HASH or ARRAY element or slice

(F) The argument to `delete()` must be either a hash or array element, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
```

or a hash or array slice, such as:

```
@foo[$bar, $baz, $xyzzy]
@{$ref->[12]}{"susie", "queue"}
```

%s argument is not a subroutine name

(F) The argument to `exists()` for `exists &sub` must be a subroutine name, and not a subroutine call. `exists &sub()` will generate this error.

%s package attribute may clash with future reserved word: %s

(W reserved) A lowercase attribute name was used that had a package-specific handler. That name might have a meaning to Perl itself some day, even though it doesn't yet. Perhaps you should use a mixed-case attribute name, instead. See [attributes](#).

(in cleanup) %s

(W misc) This prefix usually indicates that a `DESTROY()` method raised the indicated exception. Since destructors are usually called by the system at arbitrary points during execution, and often a vast number of times, the warning is issued only once for any number of failures that would otherwise result in the same message being repeated.

Failure of user callbacks dispatched using the `G_KEEPPERR` flag could also result in this warning. See [G_KEEPPERR](#).

< should be quotes

(F) You wrote `< require <file` when you should have written `require 'file'`.

Attempt to join self

(F) You tried to join a thread from within itself, which is an impossible task. You may be joining the wrong thread, or you may need to move the `join()` to some other thread.

Bad eval'd substitution pattern

(F) You've used the `/e` switch to evaluate the replacement for a substitution, but perl found a syntax error in the code to evaluate, most likely an unexpected right brace `'}'`.

Bad `realloc()` ignored

(S) An internal routine called `realloc()` on something that had never been `malloc()`ed in the first place. Mandatory, but can be disabled by setting environment variable `PERL_BADFREE` to 1.

Bareword found in conditional

(W bareword) The compiler found a bareword where it expected a conditional, which often indicates that an `||` or `&&` was parsed as part of the last argument of the previous construct, for example:

```
open FOO || die;
```

It may also indicate a misspelled constant that has been interpreted as a bareword:

```
use constant TYPO => 1;
if (TYOP) { print "foo" }
```

The `strict` pragma is useful in avoiding such errors.

Binary number 0b11111111111111111111111111111111 non-portable

(W portable) The binary number you specified is larger than $2^{32}-1$ (4294967295) and therefore non-portable between systems. See [perlport](#) for more on portability concerns.

Bit vector size 32 non-portable

(W portable) Using bit vector sizes larger than 32 is non-portable.

Buffer overflow in `prime_env_iter`: %s

(W internal) A warning peculiar to VMS. While Perl was preparing to iterate over %ENV, it encountered a logical name or symbol definition which was too long, so it was truncated to the string shown.

Can't check filesystem of script "%s"

(P) For some reason you can't check the filesystem of the script for `nosuid`.

Can't declare class for non-scalar %s in "%s"

(S) Currently, only scalar variables can be declared with a specific class qualifier in a "my" or "our" declaration. The semantics may be extended for other types of variables in future.

Can't declare %s in "%s"

(F) Only scalar, array, and hash variables may be declared as "my" or "our" variables. They must have ordinary identifiers as names.

Can't ignore signal CHLD, forcing to default

(W signal) Perl has detected that it is being run with the `SIGCHLD` signal (sometimes known as `SIGCLD`) disabled. Since disabling this signal will interfere with proper determination of exit status of child processes, Perl has reset the signal to its default value. This situation typically indicates that the parent program under which Perl may be running (e.g., `cron`) is being very careless.

Can't modify non-lvalue subroutine call

(F) Subroutines meant to be used in lvalue context should be declared as such, see [Lvalue subroutines in perlsub](#).

Can't read CRTL environ

(S) A warning peculiar to VMS. Perl tried to read an element of %ENV from the CRTL's internal environment array and discovered the array was missing. You need to figure out where your CRTL misplaced its environ or define `PERL_ENV_TABLES` (see [perlvms](#)) so that environ is not searched.

Can't remove %s: %s, skipping file

(S) You requested an inplace edit without creating a backup file. Perl was unable to remove the original file to replace it with the modified file. The file was left unmodified.

Can't return %s from lvalue subroutine

(F) Perl detected an attempt to return illegal lvalues (such as temporary or readonly values) from a subroutine used as an lvalue. This is not allowed.

Can't weaken a nonreference

(F) You attempted to weaken something that was not a reference. Only references can be weakened.

Character class [%s:] unknown

(F) The class in the character class [%s:] syntax is unknown. See [perlre](#).

Character class syntax [%s] belongs inside character classes

(W unsafe) The character class constructs `[:]`, `[=]`, and `[.]` go *inside* character classes, the `[]` are part of the construct, for example: `/[012[:alpha:]]345]/`. Note that `[=]` and `[.]` are not currently implemented; they are simply placeholders for future extensions.

Constant is not %s reference

(F) A constant value (perhaps declared using the `use constant` pragma) is being dereferenced, but it amounts to the wrong type of reference. The message indicates the type of reference that was expected. This usually indicates a syntax error in dereferencing the constant value. See [Constant Functions in perlsub](#) and [constant](#).

constant(%s): %s

(F) The parser found inconsistencies either while attempting to define an overloaded constant, or when trying to find the character name specified in the `\N{...}` escape. Perhaps you forgot to load the corresponding `overload` or `charnames` pragma? See [charnames](#) and [overload](#).

CORE::%s is not a keyword

(F) The `CORE::` namespace is reserved for Perl keywords.

defined(@array) is deprecated

(D) `defined()` is not usually useful on arrays because it checks for an undefined *scalar* value. If you want to see if the array is empty, just use `if (@array) { # not empty }` for example.

defined(%hash) is deprecated

(D) `defined()` is not usually useful on hashes because it checks for an undefined *scalar* value. If you want to see if the hash is empty, just use `if (%hash) { # not empty }` for example.

Did not produce a valid header

See Server error.

(Did you mean "local" instead of "our"?)

(W misc) Remember that "our" does not localize the declared global variable. You have declared it again in the same lexical scope, which seems superfluous.

Document contains no data

See Server error.

entering effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

false [] range "%s" in regexp

(W regexp) A character class range must start and end at a literal character, not another character class like `\d` or `[:alpha:]`. The `"-"` in your false range is interpreted as a literal `"-"`. Consider quoting the `"-"`, `"\"`. See [perlre](#).

Filehandle %s opened only for output

(W io) You tried to read from a filehandle opened only for writing. If you intended it to be a read/write filehandle, you needed to open it with `"+<"` or `"+"` or `"+">` instead of with `"<"` or nothing. If you intended only to read from the file, use `"<"`. See [open](#).

flock() on closed filehandle %s

(W closed) The filehandle you're attempting to `flock()` got itself closed some time before now. Check your logic flow. `flock()` operates on filehandles. Are you attempting to call `flock()` on a dirhandle by the same name?

Global symbol "%s" requires explicit package name

(F) You've said "use strict vars", which indicates that all variables must either be lexically scoped (using "my"), declared beforehand using "our", or explicitly qualified to say which package the global

variable is in (using "::").

Hexadecimal number 0xffffffff non-portable

(W portable) The hexadecimal number you specified is larger than $2^{32}-1$ (4294967295) and therefore non-portable between systems. See [perlport](#) for more on portability concerns.

Ill-formed CRTL environ value "%s"

(W internal) A warning peculiar to VMS. Perl tried to read the CRTL's internal environ array, and encountered an element without the = delimiter used to separate keys from values. The element is ignored.

Ill-formed message in prime_env_iter: [%s]

(W internal) A warning peculiar to VMS. Perl tried to read a logical name or CLI symbol definition when preparing to iterate over %ENV, and didn't see the expected delimiter between key and value, so the line was ignored.

Illegal binary digit %s

(F) You used a digit other than 0 or 1 in a binary number.

Illegal binary digit %s ignored

(W digit) You may have tried to use a digit other than 0 or 1 in a binary number. Interpretation of the binary number stopped before the offending digit.

Illegal number of bits in vec

(F) The number of bits in `vec()` (the third argument) must be a power of two from 1 to 32 (or 64, if your platform supports that).

Integer overflow in %s number

(W overflow) The hexadecimal, octal or binary number you have specified either as a literal or as an argument to `hex()` or `oct()` is too big for your architecture, and has been converted to a floating point number. On a 32-bit architecture the largest hexadecimal, octal or binary number representable without overflow is 0xFFFFFFFF, 037777777777, or 0b11111111111111111111111111111111 respectively. Note that Perl transparently promotes all numbers to a floating point representation internally—subject to loss of precision errors in subsequent operations.

Invalid %s attribute: %s

The indicated attribute for a subroutine or variable was not recognized by Perl or by a user-supplied handler. See [attributes](#).

Invalid %s attributes: %s

The indicated attributes for a subroutine or variable were not recognized by Perl or by a user-supplied handler. See [attributes](#).

invalid [] range "%s" in regexp

The offending range is now explicitly displayed.

Invalid separator character %s in attribute list

(F) Something other than a colon or whitespace was seen between the elements of an attribute list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon. See [attributes](#).

Invalid separator character %s in subroutine attribute list

(F) Something other than a colon or whitespace was seen between the elements of a subroutine attribute list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon.

leaving effective %s failed

(F) While under the use `filetest` pragma, switching the real and effective uids or gids failed.

Lvalue subs returning %s not implemented yet

(F) Due to limitations in the current implementation, array and hash values cannot be returned in subroutines used in lvalue context. See [Lvalue subroutines in perlsub](#).

Method %s not permitted

See Server error.

Missing %sbrace%s on \N{}

(F) Wrong syntax of character name literal `\N{ charname }` within double-quotish context.

Missing command in piped open

(W pipe) You used the `open (FH, " | command")` or `open (FH, "command |")` construction, but the command was missing or blank.

Missing name in "my sub"

(F) The reserved syntax for lexically scoped subroutines requires that they have a name with which they can be found.

No %s specified for -%c

(F) The indicated command line switch needs a mandatory argument, but you haven't specified one.

No package name allowed for variable %s in "our"

(F) Fully qualified variable names are not allowed in "our" declarations, because that doesn't make much sense under existing semantics. Such syntax is reserved for future extensions.

No space allowed after -%c

(F) The argument to the indicated command line switch must follow immediately after the switch, without intervening spaces.

no UTC offset information; assuming local time is UTC

(S) A warning peculiar to VMS. Perl was unable to find the local timezone offset, so it's assuming that local system time is equivalent to UTC. If it's not, define the logical name `SYS$TIMEZONE_DIFFERENTIAL` to translate to the number of seconds which need to be added to UTC to get local time.

Octal number 03777777777 non-portable

(W portable) The octal number you specified is larger than $2^{32}-1$ (4294967295) and therefore non-portable between systems. See [perlport](#) for more on portability concerns.

See also [perlport](#) for writing portable code.

panic: del_backref

(P) Failed an internal consistency check while trying to reset a weak reference.

panic: kid popen errno read

(F) forked child returned an incomprehensible message about its errno.

panic: magic_killbackrefs

(P) Failed an internal consistency check while trying to reset all weak references to an object.

Parentheses missing around "%s" list

(W parenthesis) You said something like

```
my $foo, $bar = @_;
```

when you meant


```
my ($foo, $bar) = @_;
```

Remember that "my", "our", and "local" bind tighter than comma.

Possible unintended interpolation of %s in string

(W ambiguous) It used to be that Perl would try to guess whether you wanted an array interpolated or a literal @. It no longer does this; arrays are now *always* interpolated into strings. This means that if you try something like:

```
print "fred@example.com";
```

and the array @example doesn't exist, Perl is going to print fred.com, which is probably not what you wanted. To get a literal @ sign in a string, put a backslash before it, just as you would to get a literal \$ sign.

Possible Y2K bug: %s

(W y2k) You are concatenating the number 19 with another number, which could be a potential Year 2000 problem.

pragma "attrs" is deprecated, use "sub NAME : ATTRS" instead

(W deprecated) You have written something like this:

```
sub doit
{
    use attrs qw(locked);
}
```

You should use the new declaration syntax instead.

```
sub doit : locked
{
    ...
}
```

The use attrs pragma is now obsolete, and is only provided for backward-compatibility. See [Subroutine Attributes in perlsub](#).

Premature end of script headers

See Server error.

Repeat count in pack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See [pack](#).

Repeat count in unpack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See [unpack](#).

realloc() of freed memory ignored

(S) An internal routine called realloc() on something that had already been freed.

Reference is already weak

(W misc) You have attempted to weaken a reference that is already weak. Doing so has no effect.

setpgrp can't take arguments

(F) Your system has the setpgrp() from BSD 4.2, which takes no arguments, unlike POSIX setpgid(), which takes a process ID and process group ID.

Strange *+?{} on zero-length expression

(W regexp) You applied a regular expression quantifier in a place where it makes no sense, such as on a zero-width assertion. Try putting the quantifier inside the assertion instead. For example, the way to match "abc" provided that it is followed by three repetitions of "xyz" is /abc(?:xyz){3}/, not /abc(?:xyz){3}/.

switching effective %s is not implemented

(F) While under the use `filetest` pragma, we cannot switch the real and effective uids or gids.

This Perl can't reset CRTL environ elements (%s)

This Perl can't set CRTL environ elements (%s=%s)

(W internal) Warnings peculiar to VMS. You tried to change or delete an element of the CRTL's internal environ array, but your copy of Perl wasn't built with a CRTL that contained the `setenv()` function. You'll need to rebuild Perl with a CRTL that does, or redefine ***PERL_ENV_TABLES*** (see [perlvms](#)) so that the environ array isn't the target of the change to `%ENV` which produced the warning.

Too late to run %s block

(W void) A CHECK or INIT block is being defined during run time proper, when the opportunity to run them has already passed. Perhaps you are loading a file with `require` or `do` when you should be using `use` instead. Or perhaps you should put the `require` or `do` inside a BEGIN block.

Unknown open () mode '%s'

(F) The second argument of 3-argument open () is not among the list of valid modes: < , < , << , < +< , < + , << + , - | , | - .

Unknown process %x sent message to prime_env_iter: %s

(P) An error peculiar to VMS. Perl was reading values for %ENV before iterating over it, and someone else stuck a message in the stream of data Perl expected. Someone's very confused, or perhaps trying to subvert Perl's population of %ENV for nefarious purposes.

Unrecognized escape \\%c passed through

(W misc) You used a backslash–character combination which is not recognized by Perl. The character was understood literally.

Unterminated attribute parameter in attribute list

(F) The lexer saw an opening (left) parenthesis character while parsing an attribute list, but the matching closing (right) parenthesis character was not found. You may need to add (or remove) a backslash character to get your parentheses to balance. See [attributes](#).

Unterminated attribute list

(F) The lexer found something other than a simple identifier at the start of an attribute, and it wasn't a semicolon or the start of a block. Perhaps you terminated the parameter list of the previous attribute too soon. See [attributes](#).

Unterminated attribute parameter in subroutine attribute list

(F) The lexer saw an opening (left) parenthesis character while parsing a subroutine attribute list, but the matching closing (right) parenthesis character was not found. You may need to add (or remove) a backslash character to get your parentheses to balance.

Unterminated subroutine attribute list

(F) The lexer found something other than a simple identifier at the start of a subroutine attribute, and it wasn't a semicolon or the start of a block. Perhaps you terminated the parameter list of the previous attribute too soon.

Value of CLI symbol "%s" too long

(W misc) A warning peculiar to VMS. Perl tried to read the value of an %ENV element from a CLI symbol table, and found a resultant string longer than 1024 characters. The return value has been truncated to 1024 characters.

Version number must be a constant number

(P) The attempt to translate a use `Module n.n LIST` statement into its equivalent `BEGIN` block found an internal inconsistency with the version number.

New tests**lib/attrs**

Compatibility tests for `sub : attrs` vs the older use `attrs`.

lib/env

Tests for new environment scalar capability (e.g., use `Env qw($BAR);`).

lib/env-array

Tests for new environment array capability (e.g., use `Env qw(@PATH);`).

lib/io_const

IO constants (`SEEK_*`, `_IO*`).

lib/io_dir

Directory-related IO methods (`new`, `read`, `close`, `rewind`, `tied delete`).

lib/io_multihomed

INET sockets with multi-homed hosts.

lib/io_poll

`IO poll()`.

lib/io_unix

UNIX sockets.

op/attrs

Regression tests for `my ($x, @y, %z) : attrs` and `<sub : attrs`.

op/filetest

File test operators.

op/lex_assign

Verify operations that access pad objects (lexicals and temporaries).

op/exists_sub

Verify `exists &sub` operations.

Incompatible Changes**Perl Source Incompatibilities**

Beware that any new warnings that have been added or old ones that have been enhanced are **not** considered incompatible changes.

Since all new warnings must be explicitly requested via the `-w` switch or the `warnings` pragma, it is ultimately the programmer's responsibility to ensure that warnings are enabled judiciously.

CHECK is a new keyword

All subroutine definitions named `CHECK` are now special. See `/ "Support for CHECK blocks"` for more information.

Treatment of list slices of undef has changed

There is a potential incompatibility in the behavior of list slices that are comprised entirely of undefined values. See .

Format of `$English::PERL_VERSION` is different

The `English` module now sets `$PERL_VERSION` to `$^V` (a string value) rather than `$]` (a numeric value). This is a potential incompatibility. Send us a report via `perlbug` if you are affected by this.

See for the reasons for this change.

Literals of the form 1.2.3 parse differently

Previously, numeric literals with more than one dot in them were interpreted as a floating point number concatenated with one or more numbers. Such "numbers" are now parsed as strings composed of the specified ordinals.

For example, `print 97.98.99` used to output `97.9899` in earlier versions, but now prints `abc`.

See .

Possibly changed pseudo-random number generator

Perl programs that depend on reproducing a specific set of pseudo-random numbers may now produce different output due to improvements made to the `rand()` builtin. You can use `sh Configure -Drandfunc=rand` to obtain the old behavior.

See .

Hashing function for hash keys has changed

Even though Perl hashes are not order preserving, the apparently random order encountered when iterating on the contents of a hash is actually determined by the hashing algorithm used. Improvements in the algorithm may yield a random order that is **different** from that of previous versions, especially when iterating on hashes.

See for additional information.

`undef` fails on read only values

Using the `undef` operator on a readonly value (such as `$1`) has the same effect as assigning `undef` to the readonly value—it throws an exception.

Close-on-exec bit may be set on pipe and socket handles

Pipe and socket handles are also now subject to the close-on-exec behavior determined by the special variable `$^F`.

See .

Writing `"$$1"` to mean `"${$}1"` is unsupported

Perl 5.004 deprecated the interpretation of `$$1` and similar within interpolated strings to mean `$$. "1"`, but still allowed it.

In Perl 5.6.0 and later, `"$$1"` always means `"${$}1"`.

`delete()`, `each()`, `values()` and `\(%h)`

operate on aliases to values, not copies

`delete()`, `each()`, `values()` and hashes (e.g. `\(%h)`) in a list context return the actual values in the hash, instead of copies (as they used to in earlier versions). Typical idioms for using these constructs copy the returned values, but this can make a significant difference when creating references to the returned values. Keys in the hash are still returned as copies when iterating on a hash.

See also `/"delete()", each(), values() and hash iteration are faster"`.

`vec(EXPR,OFFSET,BITS)` enforces powers-of-two BITS

`vec()` generates a run-time error if the `BITS` argument is not a valid power-of-two integer.

Text of some diagnostic output has changed

Most references to internal Perl operations in diagnostics have been changed to be more descriptive. This may be an issue for programs that may incorrectly rely on the exact text of diagnostics for proper functioning.

`%@` has been removed

The undocumented special variable `%@` that used to accumulate "background" errors (such as those that happen in `DESTROY()`) has been removed, because it could potentially result in memory leaks.

Parenthesized `not ()` behaves like a list operator

The `not` operator now falls under the "if it looks like a function, it behaves like a function" rule.

As a result, the parenthesized form can be used with `grep` and `map`. The following construct used to be a syntax error before, but it works as expected now:

```
grep not ($_), @things;
```

On the other hand, using `not` with a literal list slice may not work. The following previously allowed construct:

```
print not (1,2,3)[0];
```

needs to be written with additional parentheses now:

```
print not ((1,2,3)[0]);
```

The behavior remains unaffected when `not` is not followed by parentheses.

Semantics of bareword prototype `(*)` have changed

The semantics of the bareword prototype `*` have changed. Perl 5.005 always coerced simple scalar arguments to a `typeglob`, which wasn't useful in situations where the subroutine must distinguish between a simple scalar and a `typeglob`. The new behavior is to not coerce bareword arguments to a `typeglob`. The value will always be visible as either a simple scalar or as a reference to a `typeglob`.

See .

Semantics of bit operators may have changed on 64-bit platforms

If your platform is either natively 64-bit or if Perl has been configured to used 64-bit integers, i.e., `$Config{ivsize}` is 8, there may be a potential incompatibility in the behavior of bitwise numeric operators (`&` `|` `^` `~` `<<` `>>`). These operators used to strictly operate on the lower 32 bits of integers in previous versions, but now operate over the entire native integral width. In particular, note that unary `~` will produce different results on platforms that have different `$Config{ivsize}`. For portability, be sure to mask off the excess bits in the result of unary `~`, e.g., `~$x & 0xffffffff`.

See .

More builtins taint their results

As described in , there may be more sources of taint in a Perl program.

To avoid these new tainting behaviors, you can build Perl with the `Configure` option `-Accflags=-DINCOMPLETE_TAINTS`. Beware that the ensuing perl binary may be insecure.

C Source Incompatibilities

PERL_POLLUTE

Release 5.005 grandfathered old global symbol names by providing preprocessor macros for extension source compatibility. As of release 5.6.0, these preprocessor definitions are not available by default. You need to explicitly compile perl with `-DPERL_POLLUTE` to get these definitions. For extensions still using the old symbols, this option can be specified via `MakeMaker`:

```
perl Makefile.PL POLLUTE=1
```

PERL_IMPLICIT_CONTEXT

This new build option provides a set of macros for all API functions such that an implicit interpreter/thread context argument is passed to every API function. As a result of this, something like `sv_setsv(foo,bar)` amounts to a macro invocation that actually translates to something like `Perl_sv_setsv(my_perl,foo,bar)`. While this is generally expected to not have any significant source compatibility issues, the difference between a macro and a real function call will need to be considered.

This means that there is a source compatibility issue as a result of this if your extensions attempt to use pointers to any of the Perl API functions.

Note that the above issue is not relevant to the default build of Perl, whose interfaces continue to match those of prior versions (but subject to the other options described here).

See *The Perl API in [perlguts](#)* for detailed information on the ramifications of building Perl with this option.

NOTE: `PERL_IMPLICIT_CONTEXT` is automatically enabled whenever Perl is built with one of `-Dusethreads`, `-Dusemultiplicity`, or both. It is not intended to be enabled by users at this time.

`PERL_POLLUTE_MALLOC`

Enabling Perl's malloc in release 5.005 and earlier caused the namespace of the system's malloc family of functions to be usurped by the Perl versions, since by default they used the same names. Besides causing problems on platforms that do not allow these functions to be cleanly replaced, this also meant that the system versions could not be called in programs that used Perl's malloc. Previous versions of Perl have allowed this behaviour to be suppressed with the `HIDEMYMALLOC` and `EMBEDMYMALLOC` preprocessor definitions.

As of release 5.6.0, Perl's malloc family of functions have default names distinct from the system versions. You need to explicitly compile perl with `-DPERL_POLLUTE_MALLOC` to get the older behaviour. `HIDEMYMALLOC` and `EMBEDMYMALLOC` have no effect, since the behaviour they enabled is now the default.

Note that these functions do **not** constitute Perl's memory allocation API. See *Memory Allocation in [perlguts](#)* for further information about that.

Compatible C Source API Changes

`PATCHLEVEL` is now `PERL_VERSION`

The cpp macros `PERL_REVISION`, `PERL_VERSION`, and `PERL_SUBVERSION` are now available by default from `perl.h`, and reflect the base revision, patchlevel, and subversion respectively. `PERL_REVISION` had no prior equivalent, while `PERL_VERSION` and `PERL_SUBVERSION` were previously available as `PATCHLEVEL` and `SUBVERSION`.

The new names cause less pollution of the **cpp** namespace and reflect what the numbers have come to stand for in common practice. For compatibility, the old names are still supported when ***patchlevel.h*** is explicitly included (as required before), so there is no source incompatibility from the change.

Binary Incompatibilities

In general, the default build of this release is expected to be binary compatible for extensions built with the 5.005 release or its maintenance versions. However, specific platforms may have broken binary compatibility due to changes in the defaults used in hints files. Therefore, please be sure to always check the platform-specific README files for any notes to the contrary.

The `usethreads` or `usemultiplicity` builds are **not** binary compatible with the corresponding builds in 5.005.

On platforms that require an explicit list of exports (AIX, OS/2 and Windows, among others), purely internal symbols such as parser functions and the run time opcodes are not exported by default. Perl 5.005 used to export all functions irrespective of whether they were considered part of the public API or not.

For the full list of public API functions, see *[perlapi](#)*.

Known Problems

Localizing a tied hash element may leak memory

As of the 5.6.1 release, there is a known leak when code such as this is executed:

```
use Tie::Hash;
tie my %tie_hash => 'Tie::StdHash';
```

```
...
local($tie_hash{Foo}) = 1; # leaks
```

Known test failures

- 64-bit builds

Subtest #15 of lib/b.t may fail under 64-bit builds on platforms such as HP-UX PA64 and Linux IA64. The issue is still being investigated.

The lib/io_multihomed test may hang in HP-UX if Perl has been configured to be 64-bit. Because other 64-bit platforms do not hang in this test, HP-UX is suspect. All other tests pass in 64-bit HP-UX. The test attempts to create and connect to "multihomed" sockets (sockets which have multiple IP addresses).

Note that 64-bit support is still experimental.

- Failure of Thread tests

The subtests 19 and 20 of lib/thr5005.t test are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures—Perl 5.005_0x has the same bugs, but didn't have these tests. (Note that support for 5.005-style threading remains experimental.)

- NEXTSTEP 3.3 POSIX test failure

In NEXTSTEP 3.3p2 the implementation of the strftime(3) in the operating system libraries is buggy: the %j format numbers the days of a month starting from zero, which, while being logical to programmers, will cause the subtests 19 to 27 of the lib/posix test may fail.

- Tru64 (aka Digital UNIX, aka DEC OSF/1) lib/sdbm test failure with gcc

If compiled with gcc 2.95 the lib/sdbm test will fail (dump core). The cure is to use the vendor cc, it comes with the operating system and produces good code.

EBCDIC platforms not fully supported

In earlier releases of Perl, EBCDIC environments like OS390 (also known as Open Edition MVS) and VM-ESA were supported. Due to changes required by the UTF-8 (Unicode) support, the EBCDIC platforms are not supported in Perl 5.6.0.

The 5.6.1 release improves support for EBCDIC platforms, but they are not fully supported yet.

UNICOS/mk CC failures during Configure run

In UNICOS/mk the following errors may appear during the Configure run:

```
Guessing which symbols your C compiler and preprocessor define...
CC-20 cc: ERROR File = try.c, Line = 3
...
bad switch yylook 79bad switch yylook 79bad switch yylook 79bad switch yylo
...
4 errors detected in the compilation of "try.c".
```

The culprit is the broken awk of UNICOS/mk. The effect is fortunately rather mild: Perl itself is not adversely affected by the error, only the h2ph utility coming with Perl, and that is rather rarely needed these days.

Arrow operator and arrays

When the left argument to the arrow operator `< -` is an array, or the `scalar` operator operating on an array, the result of the operation must be considered erroneous. For example:

```
@x->[2]
scalar(@x)->[2]
```

These expressions will get run-time errors in some future release of Perl.

Experimental features

As discussed above, many features are still experimental. Interfaces and implementation of these features are subject to change, and in extreme cases, even subject to removal in some future release of Perl. These features include the following:

- Threads
- Unicode
- 64-bit support
- Lvalue subroutines
- Weak references
- The pseudo-hash data type
- The Compiler suite
- Internal implementation of file globbing
- The DB module
- The regular expression code constructs:
 - (?`code`) and (??`code`)

Obsolete Diagnostics

Character class syntax `[:]` is reserved for future extensions

(W) Within regular expression character classes (`[]`) the syntax beginning with `[:` and ending with `:]` is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: `"\[:` and `:\:]`.

Ill-formed logical name `[%s]` in `prime_env_iter`

(W) A warning peculiar to VMS. A logical name was encountered when preparing to iterate over `%ENV` which violates the syntactic rules governing logical names. Because it cannot be translated normally, it is skipped, and will not appear in `%ENV`. This may be a benign occurrence, as some software packages might directly modify logical name tables and introduce nonstandard names, or it may indicate that a logical name table has been corrupted.

In string, `@%s` now must be written as `\@%s`

The description of this error used to say:

(Someday it will simply assume that an unbackslashed `@` interpolates an array.)

That day has come, and this fatal error has been removed. It has been replaced by a non-fatal warning instead. See [/Arrays now always interpolate into double-quoted strings](#) for details.

Probable precedence problem on `%s`

(W) The compiler found a bareword where it expected a conditional, which often indicates that an `||` or `&&` was parsed as part of the last argument of the previous construct, for example:

```
open FOO || die;
```

regex too big

(F) The current implementation of regular expressions uses shorts as address offsets within a string. Unfortunately this means that if the regular expression compiles to longer than 32767, it'll blow up. Usually when you want a regular expression this big, there is a better way to do it with multiple statements. See [perlre](#).

Use of `"${$<digit}"` to mean `"${$}<digit"` is deprecated

(D) Perl versions before 5.004 misinterpreted any type marker followed by `"$"` and a digit. For example, `"${$0}"` was incorrectly taken to mean `"${$}0"` instead of `"${$0}"`. This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "\$\$0" in a string. So Perl 5.004 still interprets "\$\$<digit" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup. There may also be information at <http://www.perl.com/>, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to `perlbug@perl.org` to be analysed by the Perl porting team.

SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

HISTORY

Written by Gurusamy Sarathy <gsar@ActiveState.com>, with many contributions from The Perl Porters.

Send omissions or corrections to <perlbug@perl.org>.

NAME

perl56delta – what’s new for perl v5.6.0

DESCRIPTION

This document describes differences between the 5.005 release and the 5.6.0 release.

Core Enhancements

Interpreter cloning, threads, and concurrency

Perl 5.6.0 introduces the beginnings of support for running multiple interpreters concurrently in different threads. In conjunction with the `perl_clone()` API call, which can be used to selectively duplicate the state of any given interpreter, it is possible to compile a piece of code once in an interpreter, clone that interpreter one or more times, and run all the resulting interpreters in distinct threads.

On the Windows platform, this feature is used to emulate `fork()` at the interpreter level. See [perlfork](#) for details about that.

This feature is still in evolution. It is eventually meant to be used to selectively clone a subroutine and data reachable from that subroutine in a separate interpreter and run the cloned subroutine in a separate thread. Since there is no shared data between the interpreters, little or no locking will be needed (unless parts of the symbol table are explicitly shared). This is obviously intended to be an easy-to-use replacement for the existing threads support.

Support for cloning interpreters and interpreter concurrency can be enabled using the `-Dusethreads` Configure option (see `win32/Makefile` for how to enable it on Windows.) The resulting perl executable will be functionally identical to one that was built with `-Dmultiplicity`, but the `perl_clone()` API call will only be available in the former.

`-Dusethreads` enables the `cpp` macro `USE_ITHREADS` by default, which in turn enables Perl source code changes that provide a clear separation between the op tree and the data it operates with. The former is immutable, and can therefore be shared between an interpreter and all of its clones, while the latter is considered local to each interpreter, and is therefore copied for each clone.

Note that building Perl with the `-Dusemultiplicity` Configure option is adequate if you wish to run multiple **independent** interpreters concurrently in different threads. `-Dusethreads` only provides the additional functionality of the `perl_clone()` API call and other support for running **cloned** interpreters concurrently.

NOTE: This is an experimental feature. Implementation details are subject to change.

Lexically scoped warning categories

You can now control the granularity of warnings emitted by perl at a finer level using the `use warnings` pragma. [warnings](#) and [perllexwarn](#) have copious documentation on this feature.

Unicode and UTF-8 support

Perl now uses UTF-8 as its internal representation for character strings. The `utf8` and `bytes` pragmas are used to control this support in the current lexical scope. See [perlunicode](#), [utf8](#) and [bytes](#) for more information.

This feature is expected to evolve quickly to support some form of I/O disciplines that can be used to specify the kind of input and output data (bytes or characters). Until that happens, additional modules from CPAN will be needed to complete the toolkit for dealing with Unicode.

NOTE: This should be considered an experimental feature. Implementation details are subject to change.

Support for interpolating named characters

The new `\N` escape interpolates named characters within strings. For example, `"Hi! \N{WHITE SMILING FACE}"` evaluates to a string with a unicode smiley face at the end.

"our" declarations

An "our" declaration introduces a value that can be best understood as a lexically scoped symbolic alias to a global variable in the package that was current where the variable was declared. This is mostly useful as an alternative to the `vars` pragma, but also provides the opportunity to introduce typing and other attributes for such variables. See [our](#).

Support for strings represented as a vector of ordinals

Literals of the form `v1.2.3.4` are now parsed as a string composed of characters with the specified ordinals. This is an alternative, more readable way to construct (possibly unicode) strings instead of interpolating characters, as in `"\x{1}\x{2}\x{3}\x{4}"`. The leading `v` may be omitted if there are more than two ordinals, so `1.2.3` is parsed the same as `v1.2.3`.

Strings written in this form are also useful to represent version "numbers". It is easy to compare such version "numbers" (which are really just plain strings) using any of the usual string comparison operators `eq`, `ne`, `lt`, `gt`, etc., or perform bitwise string operations on them using `|`, `&`, etc.

In conjunction with the new `$_V` magic variable (which contains the perl version as a string), such literals can be used as a readable way to check if you're running a particular version of Perl:

```
# this will parse in older versions of Perl also
if ($_V and $_V gt v5.6.0) {
    # new features supported
}
```

`require` and `use` also have some special magic to support such literals, but this particular usage should be avoided because it leads to misleading error messages under versions of Perl which don't support vector strings. Using a true version number will ensure correct behavior in all versions of Perl:

```
require 5.006;      # run time check for v5.6
use 5.006_001;     # compile time check for v5.6.1
```

Also, `sprintf` and `printf` support the Perl-specific format flag `%v` to print ordinals of characters in arbitrary strings:

```
printf "v%vd", $_V;          # prints current version, such as "v5.5.650"
printf "%*vX", ":", $addr;   # formats IPv6 address
printf "%*vb", " ", $bits;   # displays bitstring
```

See [Scalar value constructors in perldata](#) for additional information.

Improved Perl version numbering system

Beginning with Perl version 5.6.0, the version number convention has been changed to a "dotted integer" scheme that is more commonly found in open source projects.

Maintenance versions of v5.6.0 will be released as v5.6.1, v5.6.2 etc. The next development series following v5.6.0 will be numbered v5.7.x, beginning with v5.7.0, and the next major production release following v5.6.0 will be v5.8.0.

The English module now sets `$PERL_VERSION` to `$_V` (a string value) rather than `$]` (a numeric value). (This is a potential incompatibility. Send us a report via [perlbug](#) if you are affected by this.)

The `v1.2.3` syntax is also now legal in Perl. See [Support for strings represented as a vector of ordinals](#) for more on that.

To cope with the new versioning system's use of at least three significant digits for each version component, the method used for incrementing the subversion number has also changed slightly. We assume that versions older than v5.6.0 have been incrementing the subversion component in multiples of 10. Versions after v5.6.0

will increment them by 1. Thus, using the new notation, 5.005_03 is the "same" as v5.5.30, and the first maintenance version following v5.6.0 will be v5.6.1 (which should be read as being equivalent to a floating point value of 5.006_001 in the older format, stored in \$]) .

New syntax for declaring subroutine attributes

Formerly, if you wanted to mark a subroutine as being a method call or as requiring an automatic `lock()` when it is entered, you had to declare that with a `use attrs` pragma in the body of the subroutine. That can now be accomplished with declaration syntax, like this:

```
sub mymethod : locked method ;
...
sub mymethod : locked method {
    ...
}

sub othermethod :locked :method ;
...
sub othermethod :locked :method {
    ...
}
```

(Note how only the first `:` is mandatory, and whitespace surrounding the `:` is optional.)

AutoSplit.pm and *SelfLoader.pm* have been updated to keep the attributes with the stubs they provide. See [attributes](#).

File and directory handles can be autovivified

Similar to how constructs such as `< $x-[0]` autovivify a reference, handle constructors (`open()`, `opendir()`, `pipe()`, `socketpair()`, `sysopen()`, `socket()`, and `accept()`) now autovivify a file or directory handle if the handle passed to them is an uninitialized scalar variable. This allows the constructs such as `open(my $fh, ...)` and `open(local $fh, ...)` to be used to create filehandles that will conveniently be closed automatically when the scope ends, provided there are no other references to them. This largely eliminates the need for typeglobs when opening filehandles that must be passed around, as in the following example:

```
sub myopen {
    open my $fh, "@_"
        or die "Can't open '@_': $!";
    return $fh;
}

{
    my $f = myopen("</etc/motd");
    print <$f>;
    # $f implicitly closed here
}
```

`open()` with more than two arguments

If `open()` is passed three arguments instead of two, the second argument is used as the mode and the third argument is taken to be the file name. This is primarily useful for protecting against unintended magic behavior of the traditional two-argument form. See [open](#).

64-bit support

Any platform that has 64-bit integers either

- (1) natively as longs or ints
- (2) via special compiler flags
- (3) using long long or int64_t

is able to use "quads" (64-bit integers) as follows:

- constants (decimal, hexadecimal, octal, binary) in the code
- arguments to `oct()` and `hex()`
- arguments to `print()`, `printf()` and `sprintf()` (flag prefixes `ll`, `L`, `q`)
- printed as such
- `pack()` and `unpack()` "q" and "Q" formats
- in basic arithmetics: `+` `-` `*` `/` `%` (NOTE: operating close to the limits of the integer values may produce surprising results)
- in bit arithmetics: `&` `|` `^` `~` `<<` (NOTE: these used to be forced to be 32 bits wide but now operate on the full native width.)
- `vec()`

Note that unless you have the case (a) you will have to configure and compile Perl using the `-Duse64bitint` Configure flag.

NOTE: The Configure flags `-Duselonglong` and `-Duse64bits` have been deprecated. Use `-Duse64bitint` instead.

There are actually two modes of 64-bitness: the first one is achieved using Configure `-Duse64bitint` and the second one using Configure `-Duse64bitall`. The difference is that the first one is minimal and the second one maximal. The first works in more places than the second.

The `use64bitint` does only as much as is required to get 64-bit integers into Perl (this may mean, for example, using "long longs") while your memory may still be limited to 2 gigabytes (because your pointers could still be 32-bit). Note that the name `64bitint` does not imply that your C compiler will be using 64-bit ints (it might, but it doesn't have to): the `use64bitint` means that you will be able to have 64 bits wide scalar values.

The `use64bitall` goes all the way by attempting to switch also integers (if it can), longs (and pointers) to being 64-bit. This may create an even more binary incompatible Perl than `-Duse64bitint`: the resulting executable may not run at all in a 32-bit box, or you may have to reboot/reconfigure/rebuild your operating system to be 64-bit aware.

Natively 64-bit systems like Alpha and Cray need neither `-Duse64bitint` nor `-Duse64bitall`.

Last but not least: note that due to Perl's habit of always using floating point numbers, the quads are still not true integers. When quads overflow their limits (0...18_446_744_073_709_551_615 unsigned, -9_223_372_036_854_775_808...9_223_372_036_854_775_807 signed), they are silently promoted to floating point numbers, after which they will start losing precision (in their lower digits).

NOTE: 64-bit support is still experimental on most platforms. Existing support only covers the LP64 data model. In particular, the LLP64 data model is not yet supported. 64-bit libraries and system APIs on many platforms have not stabilized--your mileage may vary.

Large file support

If you have filesystems that support "large files" (files larger than 2 gigabytes), you may now also be able to create and access them from Perl.

NOTE: The default action is to enable large file support, if available on the platform.

If the large file support is on, and you have a `Fcntl` constant `O_LARGEFILE`, the `O_LARGEFILE` is automatically added to the flags of `sysopen()`.

Beware that unless your filesystem also supports "sparse files" seeking to umpteen petabytes may be

inadvisable.

Note that in addition to requiring a proper file system to do large files you may also need to adjust your per-process (or your per-system, or per-process-group, or per-user-group) maximum filesize limits before running Perl scripts that try to handle large files, especially if you intend to write such files.

Finally, in addition to your process/process group maximum filesize limits, you may have quota limits on your filesystems that stop you (your user id or your user group id) from using large files.

Adjusting your process/user/group/file system/operating system limits is outside the scope of Perl core language. For process limits, you may try increasing the limits using your shell's `limits/limit/ulimit` command before running Perl. The `BSD::Resource` extension (not included with the standard Perl distribution) may also be of use, it offers the `getrlimit/setrlimit` interface that can be used to adjust process resource usage limits, including the maximum filesize limit.

Long doubles

In some systems you may be able to use long doubles to enhance the range and precision of your double precision floating point numbers (that is, Perl's numbers). Use `Configure -Duselongdouble` to enable this support (if it is available).

"more bits"

You can `"Configure -Dusemorebits"` to turn on both the 64-bit support and the long double support.

Enhanced support for `sort()` subroutines

Perl subroutines with a prototype of `($$)`, and XSUBs in general, can now be used as sort subroutines. In either case, the two elements to be compared are passed as normal parameters in `@_`. See [sort](#).

For unprototyped sort subroutines, the historical behavior of passing the elements to be compared as the global variables `$a` and `$b` remains unchanged.

`sort $coderef @foo` allowed

`sort()` did not accept a subroutine reference as the comparison function in earlier versions. This is now permitted.

File globbing implemented internally

Perl now uses the `File::Glob` implementation of the `glob()` operator automatically. This avoids using an external `csh` process and the problems associated with it.

NOTE: This is currently an experimental feature. Interfaces and implementation are subject to change.

Support for CHECK blocks

In addition to `BEGIN`, `INIT`, `END`, `DESTROY` and `AUTOLOAD`, subroutines named `CHECK` are now special. These are queued up during compilation and behave similar to `END` blocks, except they are called at the end of compilation rather than at the end of execution. They cannot be called directly.

POSIX character class syntax `[:]` supported

For example to match alphabetic characters use `/[[:alpha:]]/`. See [perlre](#) for details.

Better pseudo-random number generator

In 5.005_0x and earlier, perl's `rand()` function used the C library `rand(3)` function. As of 5.005_52, `Configure` tests for `drand48()`, `random()`, and `rand()` (in that order) and picks the first one it finds.

These changes should result in better random numbers from `rand()`.

Improved `qw//` operator

The `qw//` operator is now evaluated at compile time into a true list instead of being replaced with a run time call to `split()`. This removes the confusing misbehaviour of `qw//` in scalar context, which had inherited that behaviour from `split()`.

Thus:

```
$foo = ($bar) = qw(a b c); print "$foo|$bar\n";
```

now correctly prints "3|a", instead of "2|a".

Better worst-case behavior of hashes

Small changes in the hashing algorithm have been implemented in order to improve the distribution of lower order bits in the hashed value. This is expected to yield better performance on keys that are repeated sequences.

pack() format 'Z' supported

The new format type 'Z' is useful for packing and unpacking null-terminated strings. See [pack in perlfunc](#).

pack() format modifier '!' supported

The new format type modifier '!' is useful for packing and unpacking native shorts, ints, and longs. See [pack in perlfunc](#).

pack() and unpack() support counted strings

The template character 'c' can be used to specify a counted string type to be packed or unpacked. See [pack in perlfunc](#).

Comments in pack() templates

The '#' character in a template introduces a comment up to end of the line. This facilitates documentation of pack() templates.

Weak references

In previous versions of Perl, you couldn't cache objects so as to allow them to be deleted if the last reference from outside the cache is deleted. The reference in the cache would hold a reference count on the object and the objects would never be destroyed.

Another familiar problem is with circular references. When an object references itself, its reference count would never go down to zero, and it would not get destroyed until the program is about to exit.

Weak references solve this by allowing you to "weaken" any reference, that is, make it not count towards the reference count. When the last non-weak reference to an object is deleted, the object is destroyed and all the weak references to the object are automatically undef-ed.

To use this feature, you need the Devel::WeakRef package from CPAN, which contains additional documentation.

NOTE: This is an experimental feature. Details are subject to change.

Binary numbers supported

Binary numbers are now supported as literals, in s?printf formats, and oct():

```
$answer = 0b101010;
printf "The answer is: %b\n", oct("0b101010");
```

Lvalue subroutines

Subroutines can now return modifiable lvalues. See [Lvalue subroutines in perlsub](#).

NOTE: This is an experimental feature. Details are subject to change.

Some arrows may be omitted in calls through references

Perl now allows the arrow to be omitted in many constructs involving subroutine calls through references. For example, `< $foo[10]-('foo')` may now be written `$foo[10] ('foo')`. This is rather similar to how the arrow may be omitted from `< $foo[10]-{ 'foo' }`. Note however, that the arrow is still required for `< foo(10)-('bar')`.

Boolean assignment operators are legal lvalues

Constructs such as `($a ||= 2) += 1` are now allowed.

`exists()` is supported on subroutine names

The `exists()` builtin now works on subroutine names. A subroutine is considered to exist if it has been declared (even if implicitly). See [exists](#) for examples.

`exists()` and `delete()` are supported on array elements

The `exists()` and `delete()` builtins now work on simple arrays as well. The behavior is similar to that on hash elements.

`exists()` can be used to check whether an array element has been initialized. This avoids autovivifying array elements that don't exist. If the array is tied, the `EXISTS()` method in the corresponding tied package will be invoked.

`delete()` may be used to remove an element from the array and return it. The array element at that position returns to its uninitialized state, so that testing for the same element with `exists()` will return false. If the element happens to be the one at the end, the size of the array also shrinks up to the highest element that tests true for `exists()`, or 0 if none such is found. If the array is tied, the `DELETE()` method in the corresponding tied package will be invoked.

See [exists](#) and [delete](#) for examples.

Pseudo-hashes work better

Dereferencing some types of reference values in a pseudo-hash, such as `< $ph-{foo}[1]`, was accidentally disallowed. This has been corrected.

When applied to a pseudo-hash element, `exists()` now reports whether the specified value exists, not merely if the key is valid.

`delete()` now works on pseudo-hashes. When given a pseudo-hash element or slice it deletes the values corresponding to the keys (but not the keys themselves). See

[Pseudo-hashes: Using an array as a hash in perlref](#).

Pseudo-hash slices with constant keys are now optimized to array lookups at compile-time.

List assignments to pseudo-hash slices are now supported.

The `fields` pragma now provides ways to create pseudo-hashes, via `fields::new()` and `fields::phash()`. See [fields](#).

NOTE: The pseudo-hash data type continues to be experimental. Limiting oneself to the interface elements provided by the `fields` pragma will provide protection from any future changes.

Automatic flushing of output buffers

`fork()`, `exec()`, `system()`, `qx//`, and pipe `open()`s now flush buffers of all files opened for output when the operation was attempted. This mostly eliminates confusing buffering mishaps suffered by users unaware of how Perl internally handles I/O.

This is not supported on some platforms like Solaris where a suitably correct implementation of `fflush(NULL)` isn't available.

Better diagnostics on meaningless filehandle operations

Constructs such as `< open(<FH)` and `< close(<FH)` are compile time errors. Attempting to read from filehandles that were opened only for writing will now produce warnings (just as writing to read-only filehandles does).

Where possible, buffered data discarded from duped input filehandle

`< open(NEW, "<&OLD")` now attempts to discard any data that was previously read and buffered in OLD before duping the handle. On platforms where doing this is allowed, the next read operation on NEW will return the same data as the corresponding operation on OLD. Formerly, it would have returned the data from the start of the following disk block instead.

eof() has the same old magic as <

`eof()` would return true if no attempt to read from `<` had yet been made. `eof()` has been changed to have a little magic of its own, it now opens the `<` files.

binmode() can be used to set :crlf and :raw modes

`binmode()` now accepts a second argument that specifies a discipline for the handle in question. The two pseudo-disciplines `":raw"` and `":crlf"` are currently supported on DOS-derivative platforms. See [binmode in perlfunc](#) and [open](#).

-T filetest recognizes UTF-8 encoded files as "text"

The algorithm used for the `-T` filetest has been enhanced to correctly identify UTF-8 content as "text".

system(), backticks and pipe open now reflect exec() failure

On Unix and similar platforms, `system()`, `qx()` and `open(FOO, "cmd |")` etc., are implemented via `fork()` and `exec()`. When the underlying `exec()` fails, earlier versions did not report the error properly, since the `exec()` happened to be in a different process.

The child process now communicates with the parent about the error in launching the external command, which allows these constructs to return with their usual error value and set `$!`.

Improved diagnostics

Line numbers are no longer suppressed (under most likely circumstances) during the global destruction phase.

Diagnostics emitted from code running in threads other than the main thread are now accompanied by the thread ID.

Embedded null characters in diagnostics now actually show up. They used to truncate the message in prior versions.

`$foo::a` and `$foo::b` are now exempt from "possible typo" warnings only if `sort()` is encountered in package `foo`.

Unrecognized alphabetic escapes encountered when parsing quote constructs now generate a warning, since they may take on new semantics in later versions of Perl.

Many diagnostics now report the internal operation in which the warning was provoked, like so:

```
Use of uninitialized value in concatenation (.) at (eval 1) line 1.
Use of uninitialized value in print at (eval 1) line 1.
```

Diagnostics that occur within `eval` may also report the file and line number where the `eval` is located, in addition to the `eval` sequence number and the line number within the evaluated text itself. For example:

```
Not enough arguments for scalar at (eval 4)[newlib/perl5db.pl:1411] line 2, at EO
```

Diagnostics follow STDERR

Diagnostic output now goes to whichever file the `STDERR` handle is pointing at, instead of always going to the underlying C runtime library's `stderr`.

More consistent close-on-exec behavior

On systems that support a `close-on-exec` flag on filehandles, the flag is now set for any handles created by `pipe()`, `socketpair()`, `socket()`, and `accept()`, if that is warranted by the value of `$^F` that may be in effect. Earlier versions neglected to set the flag for handles created with these operators. See [pipe](#),

socketpair, *socket*, *accept*, and $\F .

syswrite() ease-of-use

The length argument of `syswrite()` has become optional.

Better syntax checks on parenthesized unary operators

Expressions such as:

```
print defined(&foo, &bar, &baz);
print uc("foo", "bar", "baz");
undef($foo, &bar);
```

used to be accidentally allowed in earlier versions, and produced unpredictable behaviour. Some produced ancillary warnings when used in this way; others silently did the wrong thing.

The parenthesized forms of most unary operators that expect a single argument now ensure that they are not called with more than one argument, making the cases shown above syntax errors. The usual behaviour of:

```
print defined &foo, &bar, &baz;
print uc "foo", "bar", "baz";
undef $foo, &bar;
```

remains unchanged. See *perlop*.

Bit operators support full native integer width

The bit operators ($\&$ | \wedge \sim \ll) now operate on the full native integral width (the exact size of which is available in `$Config{ivsize}`). For example, if your platform is either natively 64-bit or if Perl has been configured to use 64-bit integers, these operations apply to 8 bytes (as opposed to 4 bytes on 32-bit platforms). For portability, be sure to mask off the excess bits in the result of unary \sim , e.g., `~$x & 0xffffffff`.

Improved security features

More potentially unsafe operations taint their results for improved security.

The `passwd` and `shell` fields returned by the `getpwent()`, `getpwnam()`, and `getpwuid()` are now tainted, because the user can affect their own encrypted password and login shell.

The variable modified by `shmread()`, and messages returned by `msgrcv()` (and its object-oriented interface `IPC::SysV::Msg::rcv`) are also tainted, because other untrusted processes can modify messages and shared memory segments for their own nefarious purposes.

More functional bareword prototype (*)

Bareword prototypes have been rationalized to enable them to be used to override builtins that accept barewords and interpret them in a special way, such as `require` or `do`.

Arguments prototyped as `*` will now be visible within the subroutine as either a simple scalar or as a reference to a typeglob. See *Prototypes*.

require and do may be overridden

`require` and `do` 'file' operations may be overridden locally by importing subroutines of the same name into the current package (or globally by importing them into the `CORE::GLOBAL::` namespace). Overriding `require` will also affect `use`, provided the override is visible at compile-time. See *Overriding Built-in Functions in perlsb*.

$\X variables may now have names longer than one character

Formerly, $\X was synonymous with $\$\{\backslash cX\}$, but $\XY was a syntax error. Now variable names that begin with a control character may be arbitrarily long. However, for compatibility reasons, these variables *must* be written with explicit braces, as $\$\{^XY\}$ for example. $\$\{^XYZ\}$ is synonymous with $\$\{\backslash cXYZ\}$. Variable names with more than one control character, such as $\$\{^XY^Z\}$, are illegal.

The old syntax has not changed. As before, '^X' may be either a literal control-X character or the

two-character sequence ‘caret’ plus ‘X’. When braces are omitted, the variable name stops after the control character. Thus "\$^XYZ" continues to be synonymous with "\$^X . "YZ" as before.

As before, lexical variables may not have names beginning with control characters. As before, variables whose names begin with a control character are always forced to be in package ‘main’. All such variables are reserved for future extensions, except those that begin with ^_, which may be used by user programs and are guaranteed not to acquire special meaning in any future version of Perl.

New variable \$^C reflects -c switch

\$^C has a boolean value that reflects whether perl is being run in compile-only mode (i.e. via the -c switch). Since BEGIN blocks are executed under such conditions, this variable enables perl code to determine whether actions that make sense only during normal running are warranted. See [perlvar](#).

New variable \$^V contains Perl version as a string

\$^V contains the Perl version number as a string composed of characters whose ordinals match the version numbers, i.e. v5.6.0. This may be used in string comparisons.

See `Support` for strings represented as a vector of ordinals for an example.

Optional Y2K warnings

If Perl is built with the cpp macro PERL_Y2KWARN defined, it emits optional warnings when concatenating the number 19 with another number.

This behavior must be specifically enabled when running Configure. See *INSTALL* and *README.Y2K*.

Arrays now always interpolate into double-quoted strings

In double-quoted strings, arrays now interpolate, no matter what. The behavior in earlier versions of perl 5 was that arrays would interpolate into strings if the array had been mentioned before the string was compiled, and otherwise Perl would raise a fatal compile-time error. In versions 5.000 through 5.003, the error was

```
Literal @example now requires backslash
```

In versions 5.004_01 through 5.6.0, the error was

```
In string, @example now must be written as \@example
```

The idea here was to get people into the habit of writing "fred@[example.com](#)" when they wanted a literal @ sign, just as they have always written "Give me back my \">\$5" when they wanted a literal \$ sign.

Starting with 5.6.1, when Perl now sees an @ sign in a double-quoted string, it *always* attempts to interpolate an array, regardless of whether or not the array has been used or declared already. The fatal error has been downgraded to an optional warning:

```
Possible unintended interpolation of @example in string
```

This warns you that "fred@[example.com](#)" is going to turn into fred.com if you don't backslash the @. See <http://www.plover.com/~mjd/perl/at-error.html> for more details about the history here.

Modules and Pragmata

Modules

attributes

While used internally by Perl as a pragma, this module also provides a way to fetch subroutine and variable attributes. See [attributes](#).

- B The Perl Compiler suite has been extensively reworked for this release. More of the standard Perl testsuite passes when run under the Compiler, but there is still a significant way to go to achieve production quality compiled executables.

NOTE: The Compiler suite remains highly experimental. The generated code may not be correct, even when it manages to execute

without errors.

Benchmark

Overall, Benchmark results exhibit lower average error and better timing accuracy.

You can now run tests for *n* seconds instead of guessing the right number of tests to run: e.g., `timethese(-5, ...)` will run each code for at least 5 CPU seconds. Zero as the "number of repetitions" means "for at least 3 CPU seconds". The output format has also changed. For example:

```
use Benchmark;$x=3;timethese(-5,{a=>sub{$x*$x},b=>sub{$x**2}})
```

will now output something like this:

```
Benchmark: running a, b, each for at least 5 CPU seconds...
```

```
a:  5 wallclock secs ( 5.77 usr +  0.00 sys =  5.77 CPU) @ 200551.91/
```

```
b:  4 wallclock secs ( 5.00 usr +  0.02 sys =  5.02 CPU) @ 159605.18/
```

New features: "each for at least N CPU seconds...", "wallclock secs", and the "@ operations/CPU second (n=operations)".

`timethese()` now returns a reference to a hash of Benchmark objects containing the test results, keyed on the names of the tests.

`timethis()` now returns the iterations field in the Benchmark result object instead of 0.

`timethese()`, `timethis()`, and the new `cmpthese()` (see below) can also take a format specifier of 'none' to suppress output.

A new function `countit()` is just like `timeit()` except that it takes a TIME instead of a COUNT.

A new function `cmpthese()` prints a chart comparing the results of each test returned from a `timethese()` call. For each possible pair of tests, the percentage speed difference (iters/sec or seconds/iter) is shown.

For other details, see [Benchmark](#).

ByteLoader

The ByteLoader is a dedicated extension to generate and run Perl bytecode. See [ByteLoader](#).

constant

References can now be used.

The new version also allows a leading underscore in constant names, but disallows a double leading underscore (as in `__LINE__`). Some other names are disallowed or warned against, including BEGIN, END, etc. Some names which were forced into `main::` used to fail silently in some cases; now they're fatal (outside of `main::`) and an optional warning (inside of `main::`). The ability to detect whether a constant had been set with a given name has been added.

See [constant](#).

chardnames

This pragma implements the `\N` string escape. See [chardnames](#).

Data::Dumper

A `Maxdepth` setting can be specified to avoid venturing too deeply into deep data structures. See [Data::Dumper](#).

The XSUB implementation of `Dump()` is now automatically called if the `Useqq` setting is not in use.

Dumping `qr//` objects works correctly.

DB DB is an experimental module that exposes a clean abstraction to Perl's debugging API.

DB_File

DB_File can now be built with Berkeley DB versions 1, 2 or 3. See `ext/DB_File/Changes`.

Devel::DProf

Devel::DProf, a Perl source code profiler has been added. See [Devel::DProf](#) and [dprofpp](#).

Devel::Peek

The Devel::Peek module provides access to the internal representation of Perl variables and data. It is a data debugging tool for the XS programmer.

Dumpvalue

The Dumpvalue module provides screen dumps of Perl data.

DynaLoader

DynaLoader now supports a `dl_unload_file()` function on platforms that support unloading shared objects using `dlclose()`.

Perl can also optionally arrange to unload all extension shared objects loaded by Perl. To enable this, build Perl with the Configure option `-Accflags=-DDL_UNLOAD_ALL_AT_EXIT`. (This maybe useful if you are using Apache with `mod_perl`.)

English

`$PERL_VERSION` now stands for `$_V` (a string value) rather than for `$]` (a numeric value).

Env Env now supports accessing environment variables like `PATH` as array variables.

Fcntl

More Fcntl constants added: `F_SETLK64`, `F_SETLKW64`, `O_LARGEFILE` for large file (more than 4GB) access (NOTE: the `O_LARGEFILE` is automatically added to `sysopen()` flags if large file support has been configured, as is the default), Free/Net/OpenBSD locking behaviour flags `F_FLOCK`, `F_POSIX`, Linux `F_SHLCK`, and `O_ACCMODE`: the combined mask of `O_RDONLY`, `O_WRONLY`, and `O_RDWR`. The `seek()/sysseek()` constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` are available via the `:seek` tag. The `chmod()/stat()` `S_IF*` constants and `S_IS*` functions are available via the `:mode` tag.

File::Compare

A `compare_text()` function has been added, which allows custom comparison functions. See [File::Compare](#).

File::Find

File::Find now works correctly when the `wanted()` function is either autoloaded or is a symbolic reference.

A bug that caused File::Find to lose track of the working directory when pruning top-level directories has been fixed.

File::Find now also supports several other options to control its behavior. It can follow symbolic links if the `follow` option is specified. Enabling the `no_chdir` option will make File::Find skip changing the current directory when walking directories. The `untaint` flag can be useful when running with taint checks enabled.

See [File::Find](#).

File::Glob

This extension implements BSD-style file globbing. By default, it will also be used for the internal implementation of the `glob()` operator. See [File::Glob](#).

File::Spec

New methods have been added to the File::Spec module: `devnull()` returns the name of the null device (`/dev/null` on Unix) and `tmpdir()` the name of the temp directory (normally `/tmp` on Unix). There are now also methods to convert between absolute and relative filenames: `abs2rel()` and `rel2abs()`. For compatibility with operating systems that specify volume names in file paths, the `splitpath()`, `splitdir()`, and `catdir()` methods have been added.

File::Spec::Functions

The new File::Spec::Functions module provides a function interface to the File::Spec module. Allows shorthand

```
$fullname = catfile($dir1, $dir2, $file);
```

instead of

```
$fullname = File::Spec->catfile($dir1, $dir2, $file);
```

Getopt::Long

Getopt::Long licensing has changed to allow the Perl Artistic License as well as the GPL. It used to be GPL only, which got in the way of non-GPL applications that wanted to use Getopt::Long.

Getopt::Long encourages the use of Pod::Usage to produce help messages. For example:

```
use Getopt::Long;
use Pod::Usage;
my $man = 0;
my $help = 0;
GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
pod2usage(1) if $help;
pod2usage(-exitstatus => 0, -verbose => 2) if $man;

__END__

=head1 NAME

sample - Using Getopt::Long and Pod::Usage

=head1 SYNOPSIS

sample [options] [file ...]

Options:
    -help          brief help message
    -man           full documentation

=head1 OPTIONS

=over 8

=item B<-help>

Print a brief help message and exits.

=item B<-man>

Prints the manual page and exits.

=back

=head1 DESCRIPTION

B<This program> will read the given input file(s) and do something
useful with the contents thereof.
```

=cut

See [Pod::Usage](#) for details.

A bug that prevented the non-option call-back < from being specified as the first argument has been fixed.

To specify the characters < and as option starters, use <. Note, however, that changing option starters is strongly deprecated.

- IO write() and syswrite() will now accept a single-argument form of the call, for consistency with Perl's syswrite().

You can now create a TCP-based IO::Socket::INET without forcing a connect attempt. This allows you to configure its options (like making it non-blocking) and then call connect() manually.

A bug that prevented the IO::Socket::protocol() accessor from ever returning the correct value has been corrected.

IO::Socket::connect now uses non-blocking IO instead of alarm() to do connect timeouts.

IO::Socket::accept now uses select() instead of alarm() for doing timeouts.

IO::Socket::INET-new now sets \$! correctly on failure. \$@ is still set for backwards compatibility.

- JPL Java Perl Lingo is now distributed with Perl. See [jpl/README](#) for more information.

lib use lib now weeds out any trailing duplicate entries. no lib removes all named entries.

Math::BigInt

The bitwise operations <<, <<~, &, |, and ~ are now supported on bigints.

Math::Complex

The accessor methods Re, Im, arg, abs, rho, and theta can now also act as mutators (accessor \$z-Re(), mutator \$z-Re(3)).

The class method display_format and the corresponding object method display_format, in addition to accepting just one argument, now can also accept a parameter hash. Recognized keys of a parameter hash are "style", which corresponds to the old one parameter case, and two new parameters: "format", which is a printf()-style format string (defaults usually to "%.15g", you can revert to the default by setting the format string to undef) used for both parts of a complex number, and "polar_pretty_print" (defaults to true), which controls whether an attempt is made to try to recognize small multiples and rationals of pi (2pi, pi/2) at the argument (angle) of a polar complex number.

The potentially disruptive change is that in list context both methods now *return the parameter hash*, instead of only the value of the "style" parameter.

Math::Trig

A little bit of radial trigonometry (cylindrical and spherical), radial coordinate conversions, and the great circle distance were added.

Pod::Parser, Pod::InputObjects

Pod::Parser is a base class for parsing and selecting sections of pod documentation from an input stream. This module takes care of identifying pod paragraphs and commands in the input and hands off the parsed paragraphs and commands to user-defined methods which are free to interpret or translate them as they see fit.

Pod::InputObjects defines some input objects needed by Pod::Parser, and for advanced users of Pod::Parser that need more about a command besides its name and text.

As of release 5.6.0 of Perl, Pod::Parser is now the officially sanctioned "base parser code" recommended for use by all pod2xxx translators. Pod::Text (pod2text) and Pod::Man (pod2man) have

already been converted to use Pod::Parser and efforts to convert Pod::HTML (pod2html) are already underway. For any questions or comments about pod parsing and translating issues and utilities, please use the pod-people@perl.org mailing list.

For further information, please see [Pod::Parser](#) and [Pod::InputObjects](#).

Pod::Checker, podchecker

This utility checks pod files for correct syntax, according to [perlpod](#). Obvious errors are flagged as such, while warnings are printed for mistakes that can be handled gracefully. The checklist is not complete yet. See [Pod::Checker](#).

Pod::ParseUtils, Pod::Find

These modules provide a set of gizmos that are useful mainly for pod translators.

[Pod::Find](#)/[Pod::Find](#) traverses directory structures and returns found pod files, along with their canonical names (like `File::Spec::Unix`). [Pod::ParseUtils](#)/[Pod::ParseUtils](#) contains **Pod::List** (useful for storing pod list information), **Pod::Hyperlink** (for parsing the contents of `L<>` sequences) and **Pod::Cache** (for caching information about pod files, e.g., link nodes).

Pod::Select, podselect

Pod::Select is a subclass of Pod::Parser which provides a function named "podselect()" to filter out user-specified sections of raw pod documentation from an input stream. podselect is a script that provides access to Pod::Select from other scripts to be used as a filter. See [Pod::Select](#).

Pod::Usage, pod2usage

Pod::Usage provides the function "pod2usage()" to print usage messages for a Perl script based on its embedded pod documentation. The pod2usage() function is generally useful to all script authors since it lets them write and maintain a single source (the pods) for documentation, thus removing the need to create and maintain redundant usage message text consisting of information already in the pods.

There is also a pod2usage script which can be used from other kinds of scripts to print usage messages from pods (even for non-Perl scripts with pods embedded in comments).

For details and examples, please see [Pod::Usage](#).

Pod::Text and Pod::Man

Pod::Text has been rewritten to use Pod::Parser. While pod2text() is still available for backwards compatibility, the module now has a new preferred interface. See [Pod::Text](#) for the details. The new Pod::Text module is easily subclassed for tweaks to the output, and two such subclasses (Pod::Text::Termcap for man-page-style bold and underlining using termcap information, and Pod::Text::Color for markup with ANSI color sequences) are now standard.

pod2man has been turned into a module, Pod::Man, which also uses Pod::Parser. In the process, several outstanding bugs related to quotes in section headers, quoting of code escapes, and nested lists have been fixed. pod2man is now a wrapper script around this module.

SDBM_File

An EXISTS method has been added to this module (and sdbm_exists() has been added to the underlying sdbm library), so one can now call exists on an SDBM_File tied hash and get the correct result, rather than a runtime error.

A bug that may have caused data loss when more than one disk block happens to be read from the database in a single FETCH() has been fixed.

Sys::Syslog

Sys::Syslog now uses XSUBs to access facilities from syslog.h so it no longer requires syslog.ph to exist.

Sys::Hostname

Sys::Hostname now uses XSUBs to call the C library's `gethostname()` or `uname()` if they exist.

Term::ANSIColor

Term::ANSIColor is a very simple module to provide easy and readable access to the ANSI color and highlighting escape sequences, supported by most ANSI terminal emulators. It is now included standard.

Time::Local

The `timelocal()` and `timegm()` functions used to silently return bogus results when the date fell outside the machine's integer range. They now consistently `croak()` if the date falls in an unsupported range.

Win32

The error return value in list context has been changed for all functions that return a list of values. Previously these functions returned a list with a single element `undef` if an error occurred. Now these functions return the empty list in these situations. This applies to the following functions:

```
Win32::FsType
Win32::GetOSVersion
```

The remaining functions are unchanged and continue to return `undef` on error even in list context.

The `Win32::SetLastError(ERROR)` function has been added as a complement to the `Win32::GetLastError()` function.

The new `Win32::GetFullPathName(FILENAME)` returns the full absolute pathname for `FILENAME` in scalar context. In list context it returns a two-element list containing the fully qualified directory name and the filename. See [Win32](#).

XSLoader

The XSLoader extension is a simpler alternative to DynaLoader. See [XSLoader](#).

DBM Filters

A new feature called "DBM Filters" has been added to all the DBM modules—`DB_File`, `GDBM_File`, `NDBM_File`, `ODBM_File`, and `SDBM_File`. DBM Filters add four new methods to each DBM module:

```
filter_store_key
filter_store_value
filter_fetch_key
filter_fetch_value
```

These can be used to filter key-value pairs before the pairs are written to the database or just after they are read from the database. See [perl56delta](#) for further information.

Pragmata

`use attrs` is now obsolete, and is only provided for backward-compatibility. It's been replaced by the `sub : attributes` syntax. See [Subroutine Attributes in perlsub](#) and [attributes](#).

Lexical warnings pragma, `use warnings;`, to control optional warnings. See [perllexwarn](#).

`use filetest` to control the behaviour of filetests (`-r -w ...`). Currently only one subpragma implemented, "use filetest 'access';", that uses `access(2)` or equivalent to check permissions instead of using `stat(2)` as usual. This matters in filesystems where there are ACLs (access control lists): the `stat(2)` might lie, but `access(2)` knows better.

The `open` pragma can be used to specify default disciplines for handle constructors (e.g. `open()`) and for `qx//`. The two pseudo-disciplines `:raw` and `:crlf` are currently supported on DOS-derivative platforms (i.e. where `binmode` is not a no-op). See also ["/binmode\(\)](#) can be used to set `:crlf` and

:raw modes".

Utility Changes

dprofpp

dprofpp is used to display profile data generated using `Devel::DProf`. See [dprofpp](#).

find2perl

The `find2perl` utility now uses the enhanced features of the `File::Find` module. The `-depth` and `-follow` options are supported. Pod documentation is also included in the script.

h2xs

The `h2xs` tool can now work in conjunction with `C::Scan` (available from CPAN) to automatically parse real-life header files. The `-M`, `-a`, `-k`, and `-o` options are new.

perlcc

`perlcc` now supports the C and Bytecode backends. By default, it generates output from the simple C backend rather than the optimized C backend.

Support for non-Unix platforms has been improved.

perldoc

`perldoc` has been reworked to avoid possible security holes. It will not by default let itself be run as the superuser, but you may still use the `-U` switch to try to make it drop privileges first.

The Perl Debugger

Many bug fixes and enhancements were added to *perl5db.pl*, the Perl debugger. The help documentation was rearranged. New commands include `< < ?`, `< < ?`, and `< { ?` to list out current actions, `man docpage` to run your doc viewer on some perl docset, and support for quoted options. The help information was rearranged, and should be viewable once again if you're using **less** as your pager. A serious security hole was plugged—you should immediately remove all older versions of the Perl debugger as installed in previous releases, all the way back to perl3, from your system to avoid being bitten by this.

Improved Documentation

Many of the platform-specific README files are now part of the perl installation. See [perl](#) for the complete list.

perlapi.pod

The official list of public Perl API functions.

perlboot.pod

A tutorial for beginners on object-oriented Perl.

perlcompile.pod

An introduction to using the Perl Compiler suite.

perldbfilter.pod

A howto document on using the DBM filter facility.

perldebug.pod

All material unrelated to running the Perl debugger, plus all low-level guts-like details that risked crushing the casual user of the debugger, have been relocated from the old manpage to the next entry below.

perldebguts.pod

This new manpage contains excessively low-level material not related to the Perl debugger, but slightly related to debugging Perl itself. It also contains some arcane internal details of how the debugging process works that may only be of interest to developers of Perl debuggers.

perlfork.pod

Notes on the `fork()` emulation currently available for the Windows platform.

perlfilter.pod

An introduction to writing Perl source filters.

perlhack.pod

Some guidelines for hacking the Perl source code.

perlintern.pod

A list of internal functions in the Perl source code. (List is currently empty.)

perllexwarn.pod

Introduction and reference information about lexically scoped warning categories.

perlnumber.pod

Detailed information about numbers as they are represented in Perl.

perlopentut.pod

A tutorial on using `open()` effectively.

perlreftut.pod

A tutorial that introduces the essentials of references.

perltootc.pod

A tutorial on managing class data for object modules.

perltodo.pod

Discussion of the most often wanted features that may someday be supported in Perl.

perlunicode.pod

An introduction to Unicode support features in Perl.

Performance enhancements

Simple `sort()` using `{ $a <= $b }` and the like are optimized

Many common `sort()` operations using a simple inlined block are now optimized for faster performance.

Optimized assignments to lexical variables

Certain operations in the RHS of assignment statements have been optimized to directly set the lexical variable on the LHS, eliminating redundant copying overheads.

Faster subroutine calls

Minor changes in how subroutine calls are handled internally provide marginal improvements in performance.

`delete()`, `each()`, `values()` and hash iteration are faster

The hash values returned by `delete()`, `each()`, `values()` and hashes in a list context are the actual values in the hash, instead of copies. This results in significantly better performance, because it eliminates needless copying in most situations.

Installation and Configuration Improvements

–Dusethreads means something different

The `–Dusethreads` flag now enables the experimental interpreter-based thread support by default. To get the flavor of experimental threads that was in 5.005 instead, you need to run `Configure` with `"–Dusethreads –Duse5005threads"`.

As of v5.6.0, interpreter-threads support is still lacking a way to create new threads from Perl (i.e., `use Thread;` will not work with interpreter threads). `use Thread;` continues to be available when you

specify the `-Duse5005threads` option to Configure, bugs and all.

NOTE: Support for threads continues to be an experimental feature.

Interfaces and implementation are subject to sudden and drastic changes.

New Configure flags

The following new flags may be enabled on the Configure command line by running Configure with `-Dflag`.

```
usemultiplicity
usethreads useithreads      (new interpreter threads: no Perl API yet)
usethreads use5005threads   (threads as they were in 5.005)

use64bitint                  (equal to now deprecated 'use64bits')
use64bitall

uselongdouble
usemorebits
uselargefiles
usesocks                     (only SOCKS v5 supported)
```

Threadedness and 64-bitness now more daring

The Configure options enabling the use of threads and the use of 64-bitness are now more daring in the sense that they no more have an explicit list of operating systems of known threads/64-bit capabilities. In other words: if your operating system has the necessary APIs and datatypes, you should be able just to go ahead and use them, for threads by Configure `-Dusethreads`, and for 64 bits either explicitly by Configure `-Duse64bitint` or implicitly if your system has 64-bit wide datatypes. See also ["64-bit support"](#).

Long Doubles

Some platforms have "long doubles", floating point numbers of even larger range than ordinary "doubles". To enable using long doubles for Perl's scalars, use `-Duselongdouble`.

-Dusemorebits

You can enable both `-Duse64bitint` and `-Duselongdouble` with `-Dusemorebits`. See also ["64-bit support"](#).

-Duselargefiles

Some platforms support system APIs that are capable of handling large files (typically, files larger than two gigabytes). Perl will try to use these APIs if you ask for `-Duselargefiles`.

See ["Large file support"](#) for more information.

installusrbinperl

You can use `"Configure -Uinstallusrbinperl"` which causes `installperl` to skip installing perl also as `/usr/bin/perl`. This is useful if you prefer not to modify `/usr/bin` for some reason or another but harmful because many scripts assume to find Perl in `/usr/bin/perl`.

SOCKS support

You can use `"Configure -Dusesocks"` which causes Perl to probe for the SOCKS proxy protocol library (v5, not v4). For more information on SOCKS, see:

<http://www.socks.nec.com/>

-A flag

You can "post-edit" the Configure variables using the Configure `-A` switch. The editing happens immediately after the platform specific hints files have been processed but before the actual configuration process starts. Run `Configure -h` to find out the full `-A` syntax.

Enhanced Installation Directories

The installation structure has been enriched to improve the support for maintaining multiple versions of perl, to provide locations for vendor-supplied modules, scripts, and manpages, and to ease maintenance of

locally-added modules, scripts, and manpages. See the section on Installation Directories in the INSTALL file for complete details. For most users building and installing from source, the defaults should be fine.

If you previously used `Configure -Dsitelib` or `-Dsitearch` to set special values for library directories, you might wish to consider using the new `-Dsiteprefix` setting instead. Also, if you wish to re-use a `config.sh` file from an earlier version of perl, you should be sure to check that Configure makes sensible choices for the new directories. See INSTALL for complete details.

Platform specific changes

Supported platforms

- The Mach CThreads (NEXTSTEP, OPENSTEP) are now supported by the Thread extension.
- GNU/Hurd is now supported.
- Rhapsody/Darwin is now supported.
- EPOC is now supported (on Psion 5).
- The cygwin port (formerly cygwin32) has been greatly improved.

DOS

- Perl now works with djgpp 2.02 (and 2.03 alpha).
- Environment variable names are not converted to uppercase any more.
- Incorrect exit codes from backticks have been fixed.
- This port continues to use its own builtin globbing (not `File::Glob`).

OS390 (OpenEdition MVS)

Support for this EBCDIC platform has not been renewed in this release. There are difficulties in reconciling Perl's standardization on UTF-8 as its internal representation for characters with the EBCDIC character set, because the two are incompatible.

It is unclear whether future versions will renew support for this platform, but the possibility exists.

VMS

Numerous revisions and extensions to configuration, build, testing, and installation process to accommodate core changes and VMS-specific options.

Expand `%ENV`-handling code to allow runtime mapping to logical names, CLI symbols, and CRTL environ array.

Extension of subprocess invocation code to accept filespecs as command "verbs".

Add to Perl command line processing the ability to use default file types and to recognize Unix-style `2>&1`.

Expansion of `File::Spec::VMS` routines, and integration into `ExtUtils::MM_VMS`.

Extension of `ExtUtils::MM_VMS` to handle complex extensions more flexibly.

Barewords at start of Unix-syntax paths may be treated as text rather than only as logical names.

Optional secure translation of several logical names used internally by Perl.

Miscellaneous bugfixing and porting of new core code to VMS.

Thanks are gladly extended to the many people who have contributed VMS patches, testing, and ideas.

Win32

Perl can now emulate `fork()` internally, using multiple interpreters running in different concurrent threads. This support must be enabled at build time. See [perlfork](#) for detailed information.

When given a pathname that consists only of a drivename, such as `A:`, `opendir()` and `stat()` now use the current working directory for the drive rather than the drive root.

The builtin XSUB functions in the `Win32::` namespace are documented. See [Win32](#).

`$^X` now contains the full path name of the running executable.

A `Win32::GetLongPathName()` function is provided to complement `Win32::GetFullPathName()` and `Win32::GetShortPathName()`. See [Win32](#).

`POSIX::uname()` is supported.

`system(1,...)` now returns true process IDs rather than process handles. `kill()` accepts any real process id, rather than strictly return values from `system(1,...)`.

For better compatibility with Unix, `kill(0, $pid)` can now be used to test whether a process exists.

The `Shell` module is supported.

Better support for building Perl under `command.com` in Windows 95 has been added.

Scripts are read in binary mode by default to allow `ByteLoader` (and the filter mechanism in general) to work properly. For compatibility, the `DATA` filehandle will be set to text mode if a carriage return is detected at the end of the line containing the `__END__` or `__DATA__` token; if not, the `DATA` filehandle will be left open in binary mode. Earlier versions always opened the `DATA` filehandle in text mode.

The `glob()` operator is implemented via the `File::Glob` extension, which supports `glob` syntax of the C shell. This increases the flexibility of the `glob()` operator, but there may be compatibility issues for programs that relied on the older globbing syntax. If you want to preserve compatibility with the older syntax, you might want to run perl with `-MFile::DosGlob`. For details and compatibility information, see [File::Glob](#).

Significant bug fixes

<HANDLE on empty files

With `$/` set to `undef`, "slurping" an empty file returns a string of zero length (instead of `undef`, as it used to) the first time the `HANDLE` is read after `$/` is set to `undef`. Further reads yield `undef`.

This means that the following will append "foo" to an empty file (it used to do nothing):

```
perl -0777 -pi -e 's/^/foo/' empty_file
```

The behaviour of:

```
perl -pi -e 's/^/foo/' empty_file
```

is unchanged (it continues to leave the file empty).

eval '...' improvements

Line numbers (as reflected by `caller()` and most diagnostics) within `eval '...'` were often incorrect where here documents were involved. This has been corrected.

Lexical lookups for variables appearing in `eval '...'` within functions that were themselves called within an `eval '...'` were searching the wrong place for lexicals. The lexical search now correctly ends at the subroutine's block boundary.

The use of `return` within `eval {...}` caused `$@` not to be reset correctly when no exception occurred within the `eval`. This has been fixed.

Parsing of here documents used to be flawed when they appeared as the replacement expression in `eval 's/.../.../e'`. This has been fixed.

All compilation errors are true errors

Some "errors" encountered at compile time were by necessity generated as warnings followed by eventual termination of the program. This enabled more such errors to be reported in a single run, rather than causing

a hard stop at the first error that was encountered.

The mechanism for reporting such errors has been reimplemented to queue compile-time errors and report them at the end of the compilation as true errors rather than as warnings. This fixes cases where error messages leaked through in the form of warnings when code was compiled at run time using `eval STRING`, and also allows such errors to be reliably trapped using `eval "..."`.

Implicitly closed filehandles are safer

Sometimes implicitly closed filehandles (as when they are localized, and Perl automatically closes them on exiting the scope) could inadvertently set `$?` or `$!`. This has been corrected.

Behavior of list slices is more consistent

When taking a slice of a literal list (as opposed to a slice of an array or hash), Perl used to return an empty list if the result happened to be composed of all undef values.

The new behavior is to produce an empty list if (and only if) the original list was empty. Consider the following example:

```
@a = (1, undef, undef, 2) [2, 1, 2];
```

The old behavior would have resulted in `@a` having no elements. The new behavior ensures it has three undefined elements.

Note in particular that the behavior of slices of the following cases remains unchanged:

```
@a = () [1, 2];
@a = (getpwent) [7, 0];
@a = (anything_returning_empty_list()) [2, 1, 2];
@a = @b [2, 1, 2];
@a = @c {'a', 'b', 'c'};
```

See [perldata](#).

(\ \$) prototype and \$foo{a}

A scalar reference prototype now correctly allows a hash or array element in that slot.

goto &sub and AUTOLOAD

The `goto &sub` construct works correctly when `&sub` happens to be autoloaded.

-bareword allowed under use integer

The autoquoting of barewords preceded by `-` did not work in prior versions when the `integer` pragma was enabled. This has been fixed.

Failures in DESTROY()

When code in a destructor threw an exception, it went unnoticed in earlier versions of Perl, unless someone happened to be looking in `$@` just after the point the destructor happened to run. Such failures are now visible as warnings when warnings are enabled.

Locale bugs fixed

`printf()` and `sprintf()` previously reset the numeric locale back to the default "C" locale. This has been fixed.

Numbers formatted according to the local numeric locale (such as using a decimal comma instead of a decimal dot) caused "isn't numeric" warnings, even while the operations accessing those numbers produced correct results. These warnings have been discontinued.

Memory leaks

The `eval 'return sub {...}'` construct could sometimes leak memory. This has been fixed.

Operations that aren't filehandle constructors used to leak memory when used on invalid filehandles. This has been fixed.

Constructs that modified @_ could fail to deallocate values in @_ and thus leak memory. This has been corrected.

Spurious subroutine stubs after failed subroutine calls

Perl could sometimes create empty subroutine stubs when a subroutine was not found in the package. Such cases stopped later method lookups from progressing into base packages. This has been corrected.

Taint failures under -U

When running in unsafe mode, taint violations could sometimes cause silent failures. This has been fixed.

END blocks and the -c switch

Prior versions used to run BEGIN and END blocks when Perl was run in compile-only mode. Since this is typically not the expected behavior, END blocks are not executed anymore when the -c switch is used, or if compilation fails.

See [for](#) how to run things when the compile phase ends.

Potential to leak DATA filehandles

Using the __DATA__ token creates an implicit filehandle to the file that contains the token. It is the program's responsibility to close it when it is done reading from it.

This caveat is now better explained in the documentation. See [perldata](#).

New or Changed Diagnostics

"%s" variable %s masks earlier declaration in same %s

(W misc) A "my" or "our" variable has been redeclared in the current scope or statement, effectively eliminating all access to the previous instance. This is almost always a typographical error. Note that the earlier variable will still exist until the end of the scope or until all closure referents to it are destroyed.

"my sub" not yet implemented

(F) Lexically scoped subroutines are not yet implemented. Don't try that yet.

"our" variable %s redeclared

(W misc) You seem to have already declared the same global once before in the current lexical scope.

'!' allowed only after types %s

(F) The '!' is allowed in pack () and unpack () only after certain types. See [pack](#).

/ cannot take a count

(F) You had an unpack template indicating a counted-length string, but you have also specified an explicit size for the string. See [pack](#).

/ must be followed by a, A or Z

(F) You had an unpack template indicating a counted-length string, which must be followed by one of the letters a, A or Z to indicate what sort of string is to be unpacked. See [pack](#).

/ must be followed by a*, A* or Z*

(F) You had a pack template indicating a counted-length string, Currently the only things that can have their length counted are a*, A* or Z*. See [pack](#).

/ must follow a numeric type

(F) You had an unpack template that contained a '#', but this did not follow some numeric unpack specification. See [pack](#).

/%/s/: Unrecognized escape \\%c passed through

(W regexp) You used a backslash-character combination which is not recognized by Perl. This combination appears in an interpolated variable or a '-delimited regular expression. The character was understood literally.

`/%/s/`: Unrecognized escape `\\%c` in character class passed through

(W regexp) You used a backslash–character combination which is not recognized by Perl inside character classes. The character was understood literally.

`/%/s/` should probably be written as `"%s"`

(W syntax) You have used a pattern where Perl expected to find a string, as in the first argument to `join`. Perl will treat the true or false result of matching the pattern against `$_` as the string, which is probably not what you had in mind.

`%s()` called too early to check prototype

(W prototype) You've called a function that has a prototype before the parser saw a definition or declaration for it, and Perl could not check that the call conforms to the prototype. You need to either add an early prototype declaration for the subroutine in question, or move the subroutine definition ahead of the call to get proper prototype checking. Alternatively, if you are certain that you're calling the function correctly, you may put an ampersand before the name to avoid the warning. See [perlsub](#).

`%s` argument is not a HASH or ARRAY element

(F) The argument to `exists()` must be a hash or array element, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
```

`%s` argument is not a HASH or ARRAY element or slice

(F) The argument to `delete()` must be either a hash or array element, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
```

or a hash or array slice, such as:

```
@foo[$bar, $baz, $xyzzy]
@{$ref->[12]}{"susie", "queue"}
```

`%s` argument is not a subroutine name

(F) The argument to `exists()` for `exists &sub` must be a subroutine name, and not a subroutine call. `exists &sub()` will generate this error.

`%s` package attribute may clash with future reserved word: `%s`

(W reserved) A lowercase attribute name was used that had a package-specific handler. That name might have a meaning to Perl itself some day, even though it doesn't yet. Perhaps you should use a mixed-case attribute name, instead. See [attributes](#).

(in cleanup) `%s`

(W misc) This prefix usually indicates that a `DESTROY()` method raised the indicated exception. Since destructors are usually called by the system at arbitrary points during execution, and often a vast number of times, the warning is issued only once for any number of failures that would otherwise result in the same message being repeated.

Failure of user callbacks dispatched using the `G_KEEPPERR` flag could also result in this warning. See [G_KEEPPERR](#).

`<` should be quotes

(F) You wrote `< require <file` when you should have written `require 'file'`.

Attempt to join self

(F) You tried to join a thread from within itself, which is an impossible task. You may be joining the wrong thread, or you may need to move the `join()` to some other thread.

Bad evalled substitution pattern

(F) You've used the `/e` switch to evaluate the replacement for a substitution, but perl found a syntax error in the code to evaluate, most likely an unexpected right brace `'}'`.

Bad `realloc()` ignored

(S) An internal routine called `realloc()` on something that had never been `malloc()`ed in the first place. Mandatory, but can be disabled by setting environment variable `PERL_BADFREE` to 1.

Bareword found in conditional

(W bareword) The compiler found a bareword where it expected a conditional, which often indicates that an `||` or `&&` was parsed as part of the last argument of the previous construct, for example:

```
open FOO || die;
```

It may also indicate a misspelled constant that has been interpreted as a bareword:

```
use constant TYPO => 1;
if (TYOP) { print "foo" }
```

The `strict` pragma is useful in avoiding such errors.

Binary number `0b11111111111111111111111111111111` non-portable

(W portable) The binary number you specified is larger than $2^{32}-1$ (4294967295) and therefore non-portable between systems. See [perlport](#) for more on portability concerns.

Bit vector size 32 non-portable

(W portable) Using bit vector sizes larger than 32 is non-portable.

Buffer overflow in `prime_env_iter`: %s

(W internal) A warning peculiar to VMS. While Perl was preparing to iterate over `%ENV`, it encountered a logical name or symbol definition which was too long, so it was truncated to the string shown.

Can't check filesystem of script "%s"

(P) For some reason you can't check the filesystem of the script for `nosuid`.

Can't declare class for non-scalar %s in "%s"

(S) Currently, only scalar variables can be declared with a specific class qualifier in a "my" or "our" declaration. The semantics may be extended for other types of variables in future.

Can't declare %s in "%s"

(F) Only scalar, array, and hash variables may be declared as "my" or "our" variables. They must have ordinary identifiers as names.

Can't ignore signal `CHLD`, forcing to default

(W signal) Perl has detected that it is being run with the `SIGCHLD` signal (sometimes known as `SIGCLD`) disabled. Since disabling this signal will interfere with proper determination of exit status of child processes, Perl has reset the signal to its default value. This situation typically indicates that the parent program under which Perl may be running (e.g., `cron`) is being very careless.

Can't modify non-lvalue subroutine call

(F) Subroutines meant to be used in lvalue context should be declared as such, see [Lvalue subroutines in *perlsub*](#).

Can't read `CRTL` environ

(S) A warning peculiar to VMS. Perl tried to read an element of `%ENV` from the `CRTL`'s internal environment array and discovered the array was missing. You need to figure out where your `CRTL` misplaced its `environ` or define `PERL_ENV_TABLES` (see [perlvm](#)) so that `environ` is not searched.

Can't remove %s: %s, skipping file

(S) You requested an inplace edit without creating a backup file. Perl was unable to remove the original file to replace it with the modified file. The file was left unmodified.

Can't return %s from lvalue subroutine

(F) Perl detected an attempt to return illegal lvalues (such as temporary or readonly values) from a subroutine used as an lvalue. This is not allowed.

Can't weaken a nonreference

(F) You attempted to weaken something that was not a reference. Only references can be weakened.

Character class [:%s:] unknown

(F) The class in the character class [:] syntax is unknown. See [perlre](#).

Character class syntax [%s] belongs inside character classes

(W unsafe) The character class constructs [:], [=], and [.] go *inside* character classes, the [] are part of the construct, for example: `/[012[:alpha:]]345/`. Note that [=] and [.] are not currently implemented; they are simply placeholders for future extensions.

Constant is not %s reference

(F) A constant value (perhaps declared using the `use constant` pragma) is being dereferenced, but it amounts to the wrong type of reference. The message indicates the type of reference that was expected. This usually indicates a syntax error in dereferencing the constant value. See [Constant Functions in perlsyn](#) and [constant](#).

constant(%s): %s

(F) The parser found inconsistencies either while attempting to define an overloaded constant, or when trying to find the character name specified in the `\N{...}` escape. Perhaps you forgot to load the corresponding `overload` or `charnames` pragma? See [charnames](#) and [overload](#).

CORE::%s is not a keyword

(F) The `CORE::` namespace is reserved for Perl keywords.

defined(@array) is deprecated

(D) `defined()` is not usually useful on arrays because it checks for an undefined *scalar* value. If you want to see if the array is empty, just use `if (@array) { # not empty }` for example.

defined(%hash) is deprecated

(D) `defined()` is not usually useful on hashes because it checks for an undefined *scalar* value. If you want to see if the hash is empty, just use `if (%hash) { # not empty }` for example.

Did not produce a valid header

See Server error.

(Did you mean "local" instead of "our"?)

(W misc) Remember that "our" does not localize the declared global variable. You have declared it again in the same lexical scope, which seems superfluous.

Document contains no data

See Server error.

entering effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

false [] range "%s" in regexp

(W regexp) A character class range must start and end at a literal character, not another character class like `\d` or `[:alpha:]`. The "-" in your false range is interpreted as a literal "-". Consider quoting the "-", "\-". See [perlre](#).

Filehandle %s opened only for output

(W io) You tried to read from a filehandle opened only for writing. If you intended it to be a read/write filehandle, you needed to open it with "+<" or "+" or "+" instead of with "<" or nothing. If you intended only to read from the file, use "<". See [open](#).

flock() on closed filehandle %s

(W closed) The filehandle you're attempting to `flock()` got itself closed some time before now. Check your logic flow. `flock()` operates on filehandles. Are you attempting to call `flock()` on a dirhandle by the same name?

Global symbol "%s" requires explicit package name

(F) You've said "use strict vars", which indicates that all variables must either be lexically scoped (using "my"), declared beforehand using "our", or explicitly qualified to say which package the global variable is in (using "::").

Hexadecimal number 0xffffffff non-portable

(W portable) The hexadecimal number you specified is larger than $2^{32}-1$ (4294967295) and therefore non-portable between systems. See [perlport](#) for more on portability concerns.

Ill-formed CRTL environ value "%s"

(W internal) A warning peculiar to VMS. Perl tried to read the CRTL's internal environ array, and encountered an element without the = delimiter used to separate keys from values. The element is ignored.

Ill-formed message in prime_env_iter: [%s]

(W internal) A warning peculiar to VMS. Perl tried to read a logical name or CLI symbol definition when preparing to iterate over %ENV, and didn't see the expected delimiter between key and value, so the line was ignored.

Illegal binary digit %s

(F) You used a digit other than 0 or 1 in a binary number.

Illegal binary digit %s ignored

(W digit) You may have tried to use a digit other than 0 or 1 in a binary number. Interpretation of the binary number stopped before the offending digit.

Illegal number of bits in vec

(F) The number of bits in `vec()` (the third argument) must be a power of two from 1 to 32 (or 64, if your platform supports that).

Integer overflow in %s number

(W overflow) The hexadecimal, octal or binary number you have specified either as a literal or as an argument to `hex()` or `oct()` is too big for your architecture, and has been converted to a floating point number. On a 32-bit architecture the largest hexadecimal, octal or binary number representable without overflow is 0xFFFFFFFF, 037777777777, or 0b11111111111111111111111111111111 respectively. Note that Perl transparently promotes all numbers to a floating point representation internally—subject to loss of precision errors in subsequent operations.

Invalid %s attribute: %s

The indicated attribute for a subroutine or variable was not recognized by Perl or by a user-supplied handler. See [attributes](#).

Invalid %s attributes: %s

The indicated attributes for a subroutine or variable were not recognized by Perl or by a user-supplied handler. See [attributes](#).

invalid [] range "%s" in regexp

The offending range is now explicitly displayed.

Invalid separator character %s in attribute list

(F) Something other than a colon or whitespace was seen between the elements of an attribute list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon. See [attributes](#).

Invalid separator character %s in subroutine attribute list

(F) Something other than a colon or whitespace was seen between the elements of a subroutine attribute list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon.

leaving effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

Lvalue subs returning %s not implemented yet

(F) Due to limitations in the current implementation, array and hash values cannot be returned in subroutines used in lvalue context. See [Lvalue subroutines in perlsub](#).

Method %s not permitted

See Server error.

Missing %sbrace%s on \N{}

(F) Wrong syntax of character name literal `\N{charname}` within double-quotish context.

Missing command in piped open

(W pipe) You used the `open(FH, " | command")` or `open(FH, "command |")` construction, but the command was missing or blank.

Missing name in "my sub"

(F) The reserved syntax for lexically scoped subroutines requires that they have a name with which they can be found.

No %s specified for -%c

(F) The indicated command line switch needs a mandatory argument, but you haven't specified one.

No package name allowed for variable %s in "our"

(F) Fully qualified variable names are not allowed in "our" declarations, because that doesn't make much sense under existing semantics. Such syntax is reserved for future extensions.

No space allowed after -%c

(F) The argument to the indicated command line switch must follow immediately after the switch, without intervening spaces.

no UTC offset information; assuming local time is UTC

(S) A warning peculiar to VMS. Perl was unable to find the local timezone offset, so it's assuming that local system time is equivalent to UTC. If it's not, define the logical name **`SYS$TIMEZONE_DIFFERENTIAL`** to translate to the number of seconds which need to be added to UTC to get local time.

Octal number 03777777777 non-portable

(W portable) The octal number you specified is larger than $2^{32}-1$ (4294967295) and therefore non-portable between systems. See [perlport](#) for more on portability concerns.

See also [perlport](#) for writing portable code.

panic: del_backref

(P) Failed an internal consistency check while trying to reset a weak reference.

panic: kid popen errno read

(F) forked child returned an incomprehensible message about its errno.

panic: magic_killbackrefs

(P) Failed an internal consistency check while trying to reset all weak references to an object.

Parentheses missing around "%s" list

(W parenthesis) You said something like

```
my $foo, $bar = @_;
```

when you meant

```
my ($foo, $bar) = @_;
```

Remember that "my", "our", and "local" bind tighter than comma.

Possible unintended interpolation of %s in string

(W ambiguous) It used to be that Perl would try to guess whether you wanted an array interpolated or a literal @. It no longer does this; arrays are now *always* interpolated into strings. This means that if you try something like:

```
print "fred@example.com";
```

and the array @example doesn't exist, Perl is going to print `fred.com`, which is probably not what you wanted. To get a literal @ sign in a string, put a backslash before it, just as you would to get a literal \$ sign.

Possible Y2K bug: %s

(W y2k) You are concatenating the number 19 with another number, which could be a potential Year 2000 problem.

pragma "attrs" is deprecated, use "sub NAME : ATTRS" instead

(W deprecated) You have written something like this:

```
sub doit
{
    use attrs qw(locked);
}
```

You should use the new declaration syntax instead.

```
sub doit : locked
{
    ...
}
```

The `use attrs` pragma is now obsolete, and is only provided for backward-compatibility. See [Subroutine Attributes in perlsub](#).

Premature end of script headers

See Server error.

Repeat count in pack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See [pack](#).

Repeat count in unpack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See [unpack](#).

realloc() of freed memory ignored

(S) An internal routine called `realloc()` on something that had already been freed.

Reference is already weak

(W misc) You have attempted to weaken a reference that is already weak. Doing so has no effect.

setpgrp can't take arguments

(F) Your system has the `setpgrp()` from BSD 4.2, which takes no arguments, unlike POSIX `setpgid()`, which takes a process ID and process group ID.

Strange `*+?{}` on zero-length expression

(W regexp) You applied a regular expression quantifier in a place where it makes no sense, such as on a zero-width assertion. Try putting the quantifier inside the assertion instead. For example, the way to match "abc" provided that it is followed by three repetitions of "xyz" is `/abc(?:xyz){3}/`, not `/abc(?:xyz){3}/`.

switching effective %s is not implemented

(F) While under the use `filetest` pragma, we cannot switch the real and effective uids or gids.

This Perl can't reset CRTL environ elements (%s)**This Perl can't set CRTL environ elements (%s=%s)**

(W internal) Warnings peculiar to VMS. You tried to change or delete an element of the CRTL's internal environ array, but your copy of Perl wasn't built with a CRTL that contained the `setenv()` function. You'll need to rebuild Perl with a CRTL that does, or redefine **`PERL_ENV_TABLES`** (see [perlvms](#)) so that the environ array isn't the target of the change to `%ENV` which produced the warning.

Too late to run %s block

(W void) A CHECK or INIT block is being defined during run time proper, when the opportunity to run them has already passed. Perhaps you are loading a file with `require` or `do` when you should be using `use` instead. Or perhaps you should put the `require` or `do` inside a BEGIN block.

Unknown open() mode '%s'

(F) The second argument of 3-argument `open()` is not among the list of valid modes: `<`, `<<`, `<<<`, `<+<`, `<+<<`, `<+<<+<`, `<+<<+<+<`, `<+<<+<+<+<`.

Unknown process %x sent message to prime_env_iter: %s

(P) An error peculiar to VMS. Perl was reading values for `%ENV` before iterating over it, and someone else stuck a message in the stream of data Perl expected. Someone's very confused, or perhaps trying to subvert Perl's population of `%ENV` for nefarious purposes.

Unrecognized escape `\\%c` passed through

(W misc) You used a backslash-character combination which is not recognized by Perl. The character was understood literally.

Unterminated attribute parameter in attribute list

(F) The lexer saw an opening (left) parenthesis character while parsing an attribute list, but the matching closing (right) parenthesis character was not found. You may need to add (or remove) a backslash character to get your parentheses to balance. See [attributes](#).

Unterminated attribute list

(F) The lexer found something other than a simple identifier at the start of an attribute, and it wasn't a semicolon or the start of a block. Perhaps you terminated the parameter list of the previous attribute too soon. See [attributes](#).

Unterminated attribute parameter in subroutine attribute list

(F) The lexer saw an opening (left) parenthesis character while parsing a subroutine attribute list, but the matching closing (right) parenthesis character was not found. You may need to add (or remove) a backslash character to get your parentheses to balance.

Unterminated subroutine attribute list

(F) The lexer found something other than a simple identifier at the start of a subroutine attribute, and it wasn't a semicolon or the start of a block. Perhaps you terminated the parameter list of the previous attribute too soon.

Value of CLI symbol "%s" too long

(W misc) A warning peculiar to VMS. Perl tried to read the value of an %ENV element from a CLI symbol table, and found a resultant string longer than 1024 characters. The return value has been truncated to 1024 characters.

Version number must be a constant number

(P) The attempt to translate a `use Module n.n LIST` statement into its equivalent `BEGIN` block found an internal inconsistency with the version number.

New tests**lib/attrs**

Compatibility tests for `sub : attrs` vs the older `use attrs`.

lib/env

Tests for new environment scalar capability (e.g., `use Env qw($BAR);`).

lib/env-array

Tests for new environment array capability (e.g., `use Env qw(@PATH);`).

lib/io_const

IO constants (`SEEK_*`, `_IO*`).

lib/io_dir

Directory-related IO methods (new, read, close, rewind, tied delete).

lib/io_multihomed

INET sockets with multi-homed hosts.

lib/io_poll

`IO::poll()`.

lib/io_unix

UNIX sockets.

op/attrs

Regression tests for `my ($x, @y, %z) : attrs` and `<sub : attrs`.

op/filetest

File test operators.

op/lex_assign

Verify operations that access pad objects (lexicals and temporaries).

op/exists_sub

Verify `exists &sub` operations.

Incompatible Changes

Perl Source Incompatibilities

Beware that any new warnings that have been added or old ones that have been enhanced are **not** considered incompatible changes.

Since all new warnings must be explicitly requested via the `-w` switch or the `warnings` pragma, it is ultimately the programmer's responsibility to ensure that warnings are enabled judiciously.

CHECK is a new keyword

All subroutine definitions named `CHECK` are now special. See `/\"Support for CHECK blocks\"` for more information.

Treatment of list slices of `undef` has changed

There is a potential incompatibility in the behavior of list slices that are comprised entirely of undefined values. See [.](#)

Format of `$English::PERL_VERSION` is different

The `English` module now sets `$PERL_VERSION` to `$_V` (a string value) rather than `$]` (a numeric value). This is a potential incompatibility. Send us a report via `perlbug` if you are affected by this.

See [.](#) for the reasons for this change.

Literals of the form `1.2.3` parse differently

Previously, numeric literals with more than one dot in them were interpreted as a floating point number concatenated with one or more numbers. Such "numbers" are now parsed as strings composed of the specified ordinals.

For example, `print 97.98.99` used to output `97.9899` in earlier versions, but now prints `abc`.

See [.](#)

Possibly changed pseudo-random number generator

Perl programs that depend on reproducing a specific set of pseudo-random numbers may now produce different output due to improvements made to the `rand()` builtin. You can use `sh Configure --Drandfunc=rand` to obtain the old behavior.

See [.](#)

Hashing function for hash keys has changed

Even though Perl hashes are not order preserving, the apparently random order encountered when iterating on the contents of a hash is actually determined by the hashing algorithm used. Improvements in the algorithm may yield a random order that is **different** from that of previous versions, especially when iterating on hashes.

See [.](#) for additional information.

`undef` fails on read only values

Using the `undef` operator on a readonly value (such as `$1`) has the same effect as assigning `undef` to the readonly value—it throws an exception.

Close-on-exec bit may be set on pipe and socket handles

Pipe and socket handles are also now subject to the close-on-exec behavior determined by the special variable `$_F`.

See [.](#)

Writing `"$$1"` to mean `"${$}1"` is unsupported

Perl 5.004 deprecated the interpretation of `$$1` and similar within interpolated strings to mean ``${$}` . `"1"`, but still allowed it.

In Perl 5.6.0 and later, "\$\$1" always means "\${1}" .

`delete()`, `each()`, `values()` and `\(%h)`

operate on aliases to values, not copies

`delete()`, `each()`, `values()` and hashes (e.g. `\(%h)`) in a list context return the actual values in the hash, instead of copies (as they used to in earlier versions). Typical idioms for using these constructs copy the returned values, but this can make a significant difference when creating references to the returned values. Keys in the hash are still returned as copies when iterating on a hash.

See also `/"delete()", each(), values() and hash iteration are faster"`.

`vec(EXPR,OFFSET,BITS)` enforces powers-of-two BITS

`vec()` generates a run-time error if the BITS argument is not a valid power-of-two integer.

Text of some diagnostic output has changed

Most references to internal Perl operations in diagnostics have been changed to be more descriptive. This may be an issue for programs that may incorrectly rely on the exact text of diagnostics for proper functioning.

`%@` has been removed

The undocumented special variable `%@` that used to accumulate "background" errors (such as those that happen in `DESTROY()`) has been removed, because it could potentially result in memory leaks.

Parenthesized `not()` behaves like a list operator

The `not` operator now falls under the "if it looks like a function, it behaves like a function" rule.

As a result, the parenthesized form can be used with `grep` and `map`. The following construct used to be a syntax error before, but it works as expected now:

```
grep not($_), @things;
```

On the other hand, using `not` with a literal list slice may not work. The following previously allowed construct:

```
print not (1,2,3)[0];
```

needs to be written with additional parentheses now:

```
print not ((1,2,3)[0]);
```

The behavior remains unaffected when `not` is not followed by parentheses.

Semantics of bareword prototype `(*)` have changed

The semantics of the bareword prototype `*` have changed. Perl 5.005 always coerced simple scalar arguments to a `typeglob`, which wasn't useful in situations where the subroutine must distinguish between a simple scalar and a `typeglob`. The new behavior is to not coerce bareword arguments to a `typeglob`. The value will always be visible as either a simple scalar or as a reference to a `typeglob`.

See .

Semantics of bit operators may have changed on 64-bit platforms

If your platform is either natively 64-bit or if Perl has been configured to used 64-bit integers, i.e., `$Config{ivsize}` is 8, there may be a potential incompatibility in the behavior of bitwise numeric operators (`&` `|` `^` `~` `<<` `>>`). These operators used to strictly operate on the lower 32 bits of integers in previous versions, but now operate over the entire native integral width. In particular, note that unary `~` will produce different results on platforms that have different `$Config{ivsize}`. For portability, be sure to mask off the excess bits in the result of unary `~`, e.g., `~$x & 0xffffffff`.

See .

More builtins taint their results

As described in , there may be more sources of taint in a Perl program.

To avoid these new tainting behaviors, you can build Perl with the Configure option `-Accflags=-DINCOMPLETE_TAINTS`. Beware that the ensuing perl binary may be insecure.

C Source Incompatibilities

PERL_POLLUTE

Release 5.005 grandfathered old global symbol names by providing preprocessor macros for extension source compatibility. As of release 5.6.0, these preprocessor definitions are not available by default. You need to explicitly compile perl with `-DPERL_POLLUTE` to get these definitions. For extensions still using the old symbols, this option can be specified via MakeMaker:

```
perl Makefile.PL POLLUTE=1
```

PERL_IMPLICIT_CONTEXT

This new build option provides a set of macros for all API functions such that an implicit interpreter/thread context argument is passed to every API function. As a result of this, something like `sv_setsv(foo,bar)` amounts to a macro invocation that actually translates to something like `Perl_sv_setsv(my_perl,foo,bar)`. While this is generally expected to not have any significant source compatibility issues, the difference between a macro and a real function call will need to be considered.

This means that there **is** a source compatibility issue as a result of this if your extensions attempt to use pointers to any of the Perl API functions.

Note that the above issue is not relevant to the default build of Perl, whose interfaces continue to match those of prior versions (but subject to the other options described here).

See *[The Perl API in perl guts](#)* for detailed information on the ramifications of building Perl with this option.

NOTE: `PERL_IMPLICIT_CONTEXT` is automatically enabled whenever Perl is built with one of `-Dusethreads`, `-Dusemultiplicity`, or both. It is not intended to be enabled by users at this time.

PERL_POLLUTE_MALLOC

Enabling Perl's malloc in release 5.005 and earlier caused the namespace of the system's malloc family of functions to be usurped by the Perl versions, since by default they used the same names. Besides causing problems on platforms that do not allow these functions to be cleanly replaced, this also meant that the system versions could not be called in programs that used Perl's malloc. Previous versions of Perl have allowed this behaviour to be suppressed with the `HIDEMYMALLOC` and `EMBEDMYMALLOC` preprocessor definitions.

As of release 5.6.0, Perl's malloc family of functions have default names distinct from the system versions. You need to explicitly compile perl with `-DPERL_POLLUTE_MALLOC` to get the older behaviour. `HIDEMYMALLOC` and `EMBEDMYMALLOC` have no effect, since the behaviour they enabled is now the default.

Note that these functions do **not** constitute Perl's memory allocation API. See *[Memory Allocation in perl guts](#)* for further information about that.

Compatible C Source API Changes

PATCHLEVEL is now PERL_VERSION

The cpp macros `PERL_REVISION`, `PERL_VERSION`, and `PERL_SUBVERSION` are now available by default from `perl.h`, and reflect the base revision, patchlevel, and subversion respectively. `PERL_REVISION` had no prior equivalent, while `PERL_VERSION` and `PERL_SUBVERSION` were previously available as `PATCHLEVEL` and `SUBVERSION`.

The new names cause less pollution of the **cpp** namespace and reflect what the numbers have come to stand for in common practice. For compatibility, the old names are still supported when *patchlevel.h* is explicitly included (as required before), so there is no source incompatibility from the change.

Binary Incompatibilities

In general, the default build of this release is expected to be binary compatible for extensions built with the 5.005 release or its maintenance versions. However, specific platforms may have broken binary compatibility due to changes in the defaults used in hints files. Therefore, please be sure to always check the platform-specific README files for any notes to the contrary.

The usethreads or usemultiplicity builds are **not** binary compatible with the corresponding builds in 5.005.

On platforms that require an explicit list of exports (AIX, OS/2 and Windows, among others), purely internal symbols such as parser functions and the run time opcodes are not exported by default. Perl 5.005 used to export all functions irrespective of whether they were considered part of the public API or not.

For the full list of public API functions, see [perlapi](#).

Known Problems

Thread test failures

The subtests 19 and 20 of lib/thr5005.t test are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures—Perl 5.005_0x has the same bugs, but didn't have these tests.

EBCDIC platforms not supported

In earlier releases of Perl, EBCDIC environments like OS390 (also known as Open Edition MVS) and VM-ESA were supported. Due to changes required by the UTF-8 (Unicode) support, the EBCDIC platforms are not supported in Perl 5.6.0.

In 64-bit HP-UX the lib/io_multihomed test may hang

The lib/io_multihomed test may hang in HP-UX if Perl has been configured to be 64-bit. Because other 64-bit platforms do not hang in this test, HP-UX is suspect. All other tests pass in 64-bit HP-UX. The test attempts to create and connect to "multihomed" sockets (sockets which have multiple IP addresses).

NEXTSTEP 3.3 POSIX test failure

In NEXTSTEP 3.3p2 the implementation of the strftime(3) in the operating system libraries is buggy: the %j format numbers the days of a month starting from zero, which, while being logical to programmers, will cause the subtests 19 to 27 of the lib/posix test may fail.

Tru64 (aka Digital UNIX, aka DEC OSF/1) lib/sdbm test failure with gcc

If compiled with gcc 2.95 the lib/sdbm test will fail (dump core). The cure is to use the vendor cc, it comes with the operating system and produces good code.

UNICOS/mk CC failures during Configure run

In UNICOS/mk the following errors may appear during the Configure run:

```

Guessing which symbols your C compiler and preprocessor define...
CC-20 cc: ERROR File = try.c, Line = 3
...
bad switch yylook 79bad switch yylook 79bad switch yylook 79bad switch yylo
...
4 errors detected in the compilation of "try.c".

```

The culprit is the broken awk of UNICOS/mk. The effect is fortunately rather mild: Perl itself is not adversely affected by the error, only the h2ph utility coming with Perl, and that is rather rarely needed these days.

Arrow operator and arrays

When the left argument to the arrow operator `< -` is an array, or the `scalar` operator operating on an array, the result of the operation must be considered erroneous. For example:

```
@x->[2]
scalar(@x)->[2]
```

These expressions will get run-time errors in some future release of Perl.

Experimental features

As discussed above, many features are still experimental. Interfaces and implementation of these features are subject to change, and in extreme cases, even subject to removal in some future release of Perl. These features include the following:

- Threads
- Unicode
- 64-bit support
- Lvalue subroutines
- Weak references
- The pseudo-hash data type
- The Compiler suite
- Internal implementation of file globbing
- The DB module
- The regular expression code constructs:


```
(?{ code }) and (??{ code })
```

Obsolete Diagnostics

Character class syntax `[:]` is reserved for future extensions

(W) Within regular expression character classes (`[]`) the syntax beginning with `[":]` and ending with `"]` is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: `"\[:"` and `"]\"`.

Ill-formed logical name `[%s]` in `prime_env_iter`

(W) A warning peculiar to VMS. A logical name was encountered when preparing to iterate over `%ENV` which violates the syntactic rules governing logical names. Because it cannot be translated normally, it is skipped, and will not appear in `%ENV`. This may be a benign occurrence, as some software packages might directly modify logical name tables and introduce nonstandard names, or it may indicate that a logical name table has been corrupted.

In string, `@%s` now must be written as `\@%s`

The description of this error used to say:

```
(Someday it will simply assume that an unbackslashed @
interpolates an array.)
```

That day has come, and this fatal error has been removed. It has been replaced by a non-fatal warning instead. See [/Arrays now always interpolate into double-quoted strings](#) for details.

Probable precedence problem on `%s`

(W) The compiler found a bareword where it expected a conditional, which often indicates that an `||` or `&&` was parsed as part of the last argument of the previous construct, for example:

```
open FOO || die;
```

regex too big

(F) The current implementation of regular expressions uses shorts as address offsets within a string. Unfortunately this means that if the regular expression compiles to longer than 32767, it'll blow up. Usually when you want a regular expression this big, there is a better way to do it with multiple

statements. See *perlre*.

Use of "\$\$<digit" to mean "\${}\$<digit" is deprecated

(D) Perl versions before 5.004 misinterpreted any type marker followed by "\$" and a digit. For example, "\$\$0" was incorrectly taken to mean "\${}\$0" instead of "\${\$0}". This bug is (mostly) fixed in Perl 5.004.

However, the developers of Perl 5.004 could not fix this bug completely, because at least two widely-used modules depend on the old meaning of "\$\$0" in a string. So Perl 5.004 still interprets "\$\$<digit" in the old (broken) way inside strings; but it generates this message as a warning. And in Perl 5.005, this special treatment will cease.

Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup. There may also be information at <http://www.perl.com/perl/>, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to `perlbug@perl.org` to be analysed by the Perl porting team.

SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

HISTORY

Written by Gurusamy Sarathy <gsar@activestate.com>, with many contributions from The Perl Porters.

Send omissions or corrections to <perlbug@perl.org>.

NAME

perl570delta – what's new for perl v5.7.0

DESCRIPTION

This document describes differences between the 5.6.0 release and the 5.7.0 release.

Security Vulnerability Closed

A potential security vulnerability in the optional `suidperl` component of Perl has been identified. `suidperl` is neither built nor installed by default. As of September the 2nd, 2000, the only known vulnerable platform is Linux, most likely all Linux distributions. CERT and various vendors have been alerted about the vulnerability.

The problem was caused by Perl trying to report a suspected security exploit attempt using an external program, `/bin/mail`. On Linux platforms the `/bin/mail` program had an undocumented feature which when combined with `suidperl` gave access to a root shell, resulting in a serious compromise instead of reporting the exploit attempt. If you don't have `/bin/mail`, or if you have 'safe setuid scripts', or if `suidperl` is not installed, you are safe.

The exploit attempt reporting feature has been completely removed from the Perl 5.7.0 release, so that particular vulnerability isn't there anymore. However, further security vulnerabilities are, unfortunately, always possible. The `suidperl` code is being reviewed and if deemed too risky to continue to be supported, it may be completely removed from future releases. In any case, `suidperl` should only be used by security experts who know exactly what they are doing and why they are using `suidperl` instead of some other solution such as `sudo` (see <http://www.courtesan.com/sudo/>).

Incompatible Changes

- Arrays now always interpolate into double-quoted strings: constructs like `"foo@bar"` now always assume `@bar` is an array, whether or not the compiler has seen use of `@bar`.
- The semantics of `bless(REF, REF)` were unclear and until someone proves it to make some sense, it is forbidden.
- A reference to a reference now stringify as `"REF(0x81485ec)"` instead of `"SCALAR(0x81485ec)"` in order to be more consistent with the return value of `ref()`.
- The very dusty examples in the `eg/` directory have been removed. Suggestions for new shiny examples welcome but the main issue is that the examples need to be documented, tested and (most importantly) maintained.
- The obsolete `chat2` library that should never have been allowed to escape the laboratory has been decommissioned.
- The unimplemented POSIX regex features `[[.cc.]]` and `[[=c=]]` are still recognised but now cause fatal errors. The previous behaviour of ignoring them by default and warning if requested was unacceptable since it, in a way, falsely promised that the features could be used.
- The (bogus) escape sequences `\8` and `\9` now give an optional warning ("Unrecognized escape passed through"). There is no need to `\-escape` any `\w` character.
- `lstat(FILEHANDLE)` now gives a warning because the operation makes no sense. In future releases this may become a fatal error.
- The long deprecated uppercase aliases for the string comparison operators (`EQ`, `NE`, `LT`, `LE`, `GE`, `GT`) have now been removed.
- The regular expression captured submatches (`$1`, `$2`, ...) are now more consistently unset if the match fails, instead of leaving false data lying around in them.

- The `tr//C` and `tr//U` features have been removed and will not return; the interface was a mistake. Sorry about that. For similar functionality, see `pack('U0', ...)` and `pack('C0', ...)`.

Core Enhancements

- `perl -d:Module=arg,arg,arg` now works (previously one couldn't pass in multiple arguments.)
- `my __PACKAGE__ $obj` now works.
- `no Module;` now works even if there is no "sub unimport" in the Module.
- The numerical comparison operators return `undef` if either operand is a NaN. Previously the behaviour was unspecified.
- `pack('U0a*', ...)` can now be used to force a string to UTF8.
- `prototype(\&)` is now available.
- There is now an `UNTIE` method.

Modules and Pragmata

New Modules

- `File::Temp` allows one to create temporary files and directories in an easy, portable, and secure way.
- `Storable` gives persistence to Perl data structures by allowing the storage and retrieval of Perl data to and from files in a fast and compact binary format.

Updated And Improved Modules and Pragmata

- The following independently supported modules have been updated to newer versions from CPAN: `CGI`, `CPAN`, `DB_File`, `File::Spec`, `Getopt::Long`, the `podlators` bundle, `Pod::LaTeX`, `Pod::Parser`, `Term::ANSIColor`, `Test`.
- Bug fixes and minor enhancements have been applied to `B::Deparse`, `Data::Dumper`, `IO::Poll`, `IO::Socket::INET`, `Math::BigFloat`, `Math::Complex`, `Math::Trig`, `Net::protoent`, the `re` pragma, `SelfLoader`, `Sys::SysLog`, `Test::Harness`, `Text::Wrap`, `UNIVERSAL`, and the `warnings` pragma.
- The `attributes::reftype()` now works on tied arguments.
- `AutoLoader` can now be disabled with `no AutoLoader;`,
- The `English` module can now be used without the infamous performance hit by saying


```
use English '-no_performance_hit';
```

 (Assuming, of course, that one doesn't need the troublesome variables `$'`, `$&`, or `$'`.) Also, introduced `@LAST_MATCH_START` and `@LAST_MATCH_END` English aliases for `@-` and `@+`.
- `File::Find` now has pre- and post-processing callbacks. It also correctly changes directories when chasing symbolic links. Callbacks (naughtily) exiting with "next;" instead of "return;" now work.
- `File::Glob::glob()` renamed to `File::Glob::bsd_glob()` to avoid prototype mismatch with `CORE::glob()`.
- `IPC::Open3` now allows the use of numeric file descriptors.
- `use lib` now works identically to `@INC`. Removing directories with 'no lib' now works.
- `%INC` now localised in a Safe compartment so that `use/require` work.
- The `Shell` module now has an OO interface.

Utility Changes

- The Emacs perl mode (emacs/cperl-mode.el) has been updated to version 4.31.
- Perlbug is now much more robust. It also sends the bug report to perl.org, not perl.com.
- The perlcc utility has been rewritten and its user interface (that is, command line) is much more like that of the UNIX C compiler, cc.
- The xsubpp utility for extension writers now understands POD documentation embedded in the *.xs files.

New Documentation

- perl56delta details the changes between the 5.005 release and the 5.6.0 release.
- perldebtut is a Perl debugging tutorial.
- perlebcdic contains considerations for running Perl on EBCDIC platforms. Note that unfortunately EBCDIC platforms that used to supported back in Perl 5.005 are still unsupported by Perl 5.7.0; the plan, however, is to bring them back to the fold.
- perlnewmod tells about writing and submitting a new module.
- perlposix-bc explains using Perl on the POSIX-BC platform (an EBCDIC mainframe platform).
- perlretut is a regular expression tutorial.
- perlrequick is a regular expressions quick-start guide. Yes, much quicker than perlretut.
- perlutil explains the command line utilities packaged with the Perl distribution.

Performance Enhancements

- `map()` that changes the size of the list should now work faster.
- `sort()` has been changed to use mergesort internally as opposed to the earlier quicksort. For very small lists this may result in slightly slower sorting times, but in general the speedup should be at least 20%. Additional bonuses are that the worst case behaviour of `sort()` is now better (in computer science terms it now runs in time $O(N \log N)$, as opposed to quicksort's $\Theta(N^2)$ worst-case run time behaviour), and that `sort()` is now stable (meaning that elements with identical keys will stay ordered as they were before the sort).

Installation and Configuration Improvements

Generic Improvements

- INSTALL now explains how you can configure Perl to use 64-bit integers even on non-64-bit platforms.
- Policy.sh policy change: if you are reusing a Policy.sh file (see INSTALL) and you use `Configure -Dprefix=/foo/bar` and in the old Policy `$prefix eq $siteprefix` and `$prefix eq $vendorprefix`, all of them will now be changed to the new prefix, /foo/bar. (Previously only `$prefix` changed.) If you do not like this new behaviour, specify `prefix`, `siteprefix`, and `vendorprefix` explicitly.
- A new optional location for Perl libraries, `otherlibdirs`, is available. It can be used for example for vendor add-ons without disturbing Perl's own library directories.
- In many platforms the vendor-supplied 'cc' is too stripped-down to build Perl (basically, 'cc' doesn't do ANSI C). If this seems to be the case and 'cc' does not seem to be the GNU C compiler 'gcc', an automatic attempt is made to find and use 'gcc' instead.

- gcc needs to closely track the operating system release to avoid build problems. If Configure finds that gcc was built for a different operating system release than is running, it now gives a clearly visible warning that there may be trouble ahead.
- If binary compatibility with the 5.005 release is not wanted, Configure no longer suggests including the 5.005 modules in @INC.
- Configure -S can now run non-interactively.
- configure.gnu now works with options with whitespace in them.
- installperl now outputs everything to STDERR.
- \$Config{byteorder} is now computed dynamically (this is more robust with "fat binaries" where an executable image contains binaries for more than one binary platform.)

Selected Bug Fixes

- Several debugger fixes: exit code now reflects the script exit code, condition "0" now treated correctly, the d command now checks line number, the \$. no longer gets corrupted, all debugger output now goes correctly to the socket if RemotePort is set.
- *f{o}{FORMAT} now works.
- Lexical warnings now propagating correctly between scopes.
- Line renumbering with eval and #line now works.
- Fixed numerous memory leaks, especially in eval "".
- Modulus of unsigned numbers now works (4063328477 % 65535 used to return 27406, instead of 27047).
- Some "not a number" warnings introduced in 5.6.0 eliminated to be more compatible with 5.005. Infinity is now recognised as a number.
- our() variables will not cause "will not stay shared" warnings.
- pack "Z" now correctly terminates the string with "\0".
- Fix password routines which in some shadow password platforms (e.g. HP-UX) caused getpwent() to return every other entry.
- printf() no longer resets the numeric locale to "C".
- q(a\\b) now parses correctly as 'a\\b'.
- Printing quads (64-bit integers) with printf/sprintf now works without the q L ll prefixes (assuming you are on a quad-capable platform).
- Regular expressions on references and overloaded scalars now work.
- scalar() now forces scalar context even when used in void context.
- sort() arguments are now compiled in the right wantarray context (they were accidentally using the context of the sort() itself).
- Changed the POSIX character class [[:space:]] to include the (very rare) vertical tab character. Added a new POSIX-ish character class [[:blank:]] which stands for horizontal whitespace (currently, the space and the tab).
- \$AUTOLOAD, sort(), lock(), and spawning subprocesses in multiple threads simultaneously are now thread-safe.

- Allow read-only string on left hand side of non-modifying `tr///`.
- Several Unicode fixes (but still not perfect).
 - BOMs (byte order marks) in the beginning of Perl files (scripts, modules) should now be transparently skipped. UTF-16 (UCS-2) encoded Perl files should now be read correctly.
 - The character tables have been updated to Unicode 3.0.1.
 - `chr()` for values greater than 127 now create utf8 when under use utf8.
 - Comparing with utf8 data does not magically upgrade non-utf8 data into utf8.
 - `IsAlnum`, `IsAlpha`, and `IsWord` now match titlecase.
 - Concatenation with the `.` operator or via variable interpolation, eg, `substr`, `reverse`, `quotemeta`, the `x` operator, substitution with `s///`, single-quoted UTF8, should now work—in theory.
 - The `tr///` operator now works *slightly* better but is still rather broken. Note that the `tr///CU` functionality has been removed (but see `pack('U0', ...)`).
 - `vec()` now refuses to deal with characters 255.
 - Zero entries were missing from the Unicode classes like `IsDigit`.
- `UNIVERSAL::isa` no longer caches methods incorrectly. (This broke the Tk extension with 5.6.0.)

Platform Specific Changes and Fixes

- BSDI 4.*
Perl now works on post-4.0 BSD/OSes.
- All BSDs
Setting `$0` now works (as much as possible; see `perlvar` for details).
- Cygwin
Numerous updates; currently synchronised with Cygwin 1.1.4.
- EPOC
EPOC update after Perl 5.6.0. See `README.epoc`.
- FreeBSD 3.*
Perl now works on post-3.0 FreeBSDs.
- HP-UX
`README.hpux` updated; `Configure -Duse64bitall` now almost works.
- IRIX
Numerous compilation flag and hint enhancements; accidental mixing of 32-bit and 64-bit libraries (a doomed attempt) made much harder.
- Linux
Long doubles should now work (see `INSTALL`).
- Mac OS Classic
Compilation of the standard Perl distribution in Mac OS Classic should now work if you have the Metrowerks development environment and the missing Mac-specific toolkit bits. Contact the `macperl` mailing list for details.

- **MPE/iX**
MPE/iX update after Perl 5.6.0. See README.mpeix.
- **NetBSD/sparc**
Perl now works on NetBSD/sparc.
- **OS/2**
Now works with usethreads (see INSTALL).
- **Solaris**
64-bitness using the Sun Workshop compiler now works.
- **Tru64 (aka Digital UNIX, aka DEC OSF/1)**
The operating system version letter now recorded in `$Config{osvers}`. Allow compiling with gcc (previously explicitly forbidden). Compiling with gcc still not recommended because buggy code results, even with gcc 2.95.2.
- **Unicos**
Fixed various alignment problems that lead into core dumps either during build or later; no longer dies on math errors at runtime; now using full quad integers (64 bits), previously was using only 46 bit integers for speed.
- **VMS**
`chdir()` now works better despite a CRT bug; now works with MULTIPLICITY (see INSTALL); now works with Perl's malloc.
- **Windows**
 - `accept()` no longer leaks memory.
 - Better `chdir()` return value for a non-existent directory.
 - New `%ENV` entries now propagate to subprocesses.
 - `$ENV{LIB}` now used to search for libs under Visual C.
 - A failed (pseudo)fork now returns undef and sets `errno` to `EAGAIN`.
 - Allow `REG_EXPAND_SZ` keys in the registry.
 - Can now `send()` from all threads, not just the first one.
 - Fake signal handling reenabled, bugs and all.
 - Less stack reserved per thread so that more threads can run concurrently. (Still 16M per thread.)
 - `File::Spec-tmpdir()` now prefers `C:/temp` over `/tmp` (works better when perl is running as service).
 - Better UNC path handling under ithreads.
 - `wait()` and `waitpid()` now work much better.
 - winsock handle leak fixed.

New or Changed Diagnostics

All regular expression compilation error messages are now hopefully easier to understand both because the error message now comes before the failed regex and because the point of failure is now clearly marked.

The various "opened only for", "on closed", "never opened" warnings drop the `main::` prefix for

filehandles in the main package, for example STDIN instead of <main::STDIN.

The "Unrecognized escape" warning has been extended to include \8, \9, and _. There is no need to escape any of the \w characters.

Changed Internals

- perlapi.pod (a companion to perl guts) now attempts to document the internal API.
- You can now build a really minimal perl called microperl. Building microperl does not require even running Configure; make -f Makefile.micro should be enough. Beware: microperl makes many assumptions, some of which may be too bold; the resulting executable may crash or otherwise misbehave in wondrous ways. For careful hackers only.
- Added rsignal(), whichsig(), do_join() to the publicised API.
- Made possible to propagate customised exceptions via croak()ing.
- Added is_utf8_char(), is_utf8_string(), bytes_to_utf8(), and utf8_to_bytes().
- Now xsubs can have attributes just like subs.

Known Problems

Unicode Support Still Far From Perfect

We're working on it. Stay tuned.

EBCDIC Still A Lost Platform

The plan is to bring them back.

Building Extensions Can Fail Because Of Largefiles

Certain extensions like mod_perl and BSD::Resource are known to have issues with 'largefiles', a change brought by Perl 5.6.0 in which file offsets default to 64 bits wide, where supported. Modules may fail to compile at all or compile and work incorrectly. Currently there is no good solution for the problem, but Configure now provides appropriate non-largefile ccflags, ldflags, libswanted, and libs in the %Config hash (e.g., \$Config{ccflags_nolargefiles}) so the extensions that are having problems can try configuring themselves without the largefile-ness. This is admittedly not a clean solution, and the solution may not even work at all. One potential failure is whether one can (or, if one can, whether it's a good idea) link together at all binaries with different ideas about file offsets, all this is platform-dependent.

ftmp-security tests warn 'system possibly insecure'

Don't panic. Read INSTALL 'make test' section instead.

Test lib/posix Subtest 9 Fails In LP64-Configured HP-UX

If perl is configured with -Duse64bitall, the successful result of the subtest 10 of lib/posix may arrive before the successful result of the subtest 9, which confuses the test harness so much that it thinks the subtest 9 failed.

Long Doubles Still Don't Work In Solaris

The experimental long double support is still very much so in Solaris. (Other platforms like Linux and Tru64 are beginning to solidify in this area.)

Linux With Sflo Fails op/misc Test 48

No known fix.

Storable tests fail in some platforms

If any Storable tests fail the use of Storable is not advisable.

- Many Storable tests fail on AIX configured with 64 bit integers.

So far unidentified problems break Storable in AIX if Perl is configured to use 64 bit integers. AIX in

32-bit mode works and other 64-bit platforms work with Storable.

- DOS DJGPP may hang when testing Storable.
- st-06compat fails in UNICOS and UNICOS/mk.

This means that you cannot read old (pre-Storable-0.7) Storable images made in other platforms.

- st-store.t and st-retrieve may fail with Compaq C 6.2 on OpenVMS Alpha 7.2.

Threads Are Still Experimental

Multithreading is still an experimental feature. Some platforms emit the following message for lib/thr5005

```
#
# This is a KNOWN FAILURE, and one of the reasons why threading
# is still an experimental feature. It is here to stop people
# from deploying threads in production. ;-)
#
```

and another known thread-related warning is

```
pragma/overload.....Unbalanced saves: 3 more saves than restores
panic: magic_mutexfree during global destruction.
ok
lib/selfloader.....Unbalanced saves: 3 more saves than restores
panic: magic_mutexfree during global destruction.
ok
lib/st-dclone.....Unbalanced saves: 3 more saves than restores
panic: magic_mutexfree during global destruction.
ok
```

The Compiler Suite Is Still Experimental

The compiler suite is slowly getting better but is nowhere near working order yet. The backend part that has seen perhaps the most progress is the bytecode compiler.

Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at <http://bugs.perl.org/>. There may also be information at <http://www.perl.com/perl/>, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of perl -V, will be sent off to perlbug@perl.org to be analysed by the Perl porting team.

SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

HISTORY

Written by Jarkko Hietaniemi <jhi@iki.fi>, with many contributions from The Perl Porters and Perl Users submitting feedback and patches.

Send omissions or corrections to <perlbug@perl.org>.

NAME

perl571delta – what's new for perl v5.7.1

DESCRIPTION

This document describes differences between the 5.7.0 release and the 5.7.1 release.

(To view the differences between the 5.6.0 release and the 5.7.0 release, see [perl570delta](#).)

Security Vulnerability Closed

(This change was already made in 5.7.0 but bears repeating here.)

A potential security vulnerability in the optional `suidperl` component of Perl was identified in August 2000. `suidperl` is neither built nor installed by default. As of April 2001 the only known vulnerable platform is Linux, most likely all Linux distributions. CERT and various vendors and distributors have been alerted about the vulnerability. See <http://www.cpan.org/src/5.0/sperl-2000-08-05/sperl-2000-08-05.txt> for more information.

The problem was caused by Perl trying to report a suspected security exploit attempt using an external program, `/bin/mail`. On Linux platforms the `/bin/mail` program had an undocumented feature which when combined with `suidperl` gave access to a root shell, resulting in a serious compromise instead of reporting the exploit attempt. If you don't have `/bin/mail`, or if you have 'safe setuid scripts', or if `suidperl` is not installed, you are safe.

The exploit attempt reporting feature has been completely removed from all the Perl 5.7 releases (and will be gone also from the maintenance release 5.6.1), so that particular vulnerability isn't there anymore. However, further security vulnerabilities are, unfortunately, always possible. The `suidperl` code is being reviewed and if deemed too risky to continue to be supported, it may be completely removed from future releases. In any case, `suidperl` should only be used by security experts who know exactly what they are doing and why they are using `suidperl` instead of some other solution such as `sudo` (see <http://www.courtesan.com/sudo/>).

Incompatible Changes

- Although "you shouldn't do that", it was possible to write code that depends on Perl's hashed key order (`Data::Dumper` does this). The new algorithm "One-at-a-Time" produces a different hashed key order. More details are in .
- The list of filenames from `glob()` (or `<..`) is now by default sorted alphabetically to be `csh`-compliant. (`bsd_glob()` does still sort platform natively, ASCII or EBCDIC, unless `GLOB_ALPHASORT` is specified.)

Core Enhancements**AUTOLOAD Is Now Lvaluable**

`AUTOLOAD` is now `lvaluable`, meaning that you can add the `:lvalue` attribute to `AUTOLOAD` subroutines and you can assign to the `AUTOLOAD` return value.

PerlIO is Now The Default

- IO is now by default done via `PerlIO` rather than system's "stdio". `PerlIO` allows "layers" to be "pushed" onto a file handle to alter the handle's behaviour. Layers can be specified at open time via 3-arg form of `open`:

```
open($fh, '>:crlf :utf8', $path) || ...
```

or on already opened handles via extended `binmode`:

```
binmode($fh, ':encoding(iso-8859-7)');
```

The built-in layers are: `unix` (low level read/write), `stdio` (as in previous Perls), `perlio` (re-implementation of `stdio` buffering in a portable manner), `crlf` (does CRLF <= "\n" translation as on Win32, but available on any platform). A `mmap` layer may be available if platform supports it (mostly UNIXes).

Layers to be applied by default may be specified via the ‘open’ pragma.

See `for` for the effects of PerlIO on your architecture name.

- File handles can be marked as accepting Perl’s internal encoding of Unicode (UTF–8 or UTF–EBCDIC depending on platform) by a pseudo layer “:utf8”:

```
open($fh, ">:utf8", "Uni.txt");
```

Note for EBCDIC users: the pseudo layer “:utf8” is erroneously named for you since it’s not UTF–8 what you will be getting but instead UTF–EBCDIC. See [perlunicode](#), [utf8](#), and <http://www.unicode.org/unicode/reports/tr16/> for more information. In future releases this naming may change.

- File handles can translate character encodings from/to Perl’s internal Unicode form on read/write via the “:encoding()” layer.
- File handles can be opened to “in memory” files held in Perl scalars via:

```
open($fh, '>', \ $variable) || ...
```

- Anonymous temporary files are available without need to ‘use FileHandle’ or other module via

```
open($fh, "+>", undef) || ...
```

That is a literal undef, not an undefined value.

- The list form of `open` is now implemented for pipes (at least on UNIX):

```
open($fh, "-|", 'cat', '/etc/motd')
```

creates a pipe, and runs the equivalent of `exec('cat', '/etc/motd')` in the child process.

- The following builtin functions are now overridable: `chop()`, `chomp()`, `each()`, `keys()`, `pop()`, `push()`, `shift()`, `splice()`, `unshift()`.
- Formats now support zero–padded decimal fields.
- Perl now tries internally to use integer values in numeric conversions and basic arithmetics (+ – * /) if the arguments are integers, and tries also to keep the results stored internally as integers. This change leads into often slightly faster and always less lossy arithmetics. (Previously Perl always preferred floating point numbers in its math.)
- The `printf()` and `sprintf()` now support parameter reordering using the `%\d+\$` and `*\d+\$` syntaxes. For example

```
print "%2\$s %1\$s\n", "foo", "bar";
```

will print “bar foo\n”; This feature helps in writing internationalised software.

- Unicode in general should be now much more usable. Unicode can be used in hash keys, Unicode in regular expressions should work now, Unicode in `tr///` should work now (though `tr///` seems to be a particularly tricky to get right, so you have been warned)
- The Unicode Character Database coming with Perl has been upgraded to Unicode 3.1. For more information, see <http://www.unicode.org/>, and <http://www.unicode.org/unicode/reports/tr27/>

For developers interested in enhancing Perl’s Unicode capabilities: almost all the UCD files are included with the Perl distribution in the `lib/unicode` subdirectory. The most notable omission, for space considerations, is the UniHan database.

- The Unicode character classes `\p{Blank}` and `\p{SpacePerl}` have been added. “Blank” is like `C isblank()`, that is, it contains only “horizontal whitespace” (the space character is, the newline isn’t), and the “SpacePerl” is the Unicode equivalent of `\s` (`\p{Space}` isn’t, since that includes the vertical tabulator character, whereas `\s` doesn’t.)

Signals Are Now Safe

Perl used to be fragile in that signals arriving at inopportune moments could corrupt Perl's internal state.

Modules and Pragmata

New Modules

- `B::Concise`, by Stephen McCamant, is a new compiler backend for walking the Perl syntax tree, printing concise info about ops. The output is highly customisable.

See [B::Concise](#) for more information.

- `Class::ISA`, by Sean Burke, for reporting the search path for a class's ISA tree, has been added.

See [Class::ISA](#) for more information.

- `Cwd` has now a split personality: if possible, an extension is used, (this will hopefully be both faster and more secure and robust) but if not possible, the familiar Perl library implementation is used.
- `Digest`, a frontend module for calculating digests (checksums), from Gisle Aas, has been added.

See [Digest](#) for more information.

- `Digest::MD5` for calculating MD5 digests (checksums), by Gisle Aas, has been added.

```
use Digest::MD5 'md5_hex';

$digest = md5_hex("Thirsty Camel");

print $digest, "\n"; # 01d19d9d2045e005c3f1b80e8b164de1
```

NOTE: the MD5 backward compatibility module is deliberately not included since its use is discouraged.

See [Digest::MD5](#) for more information.

- `Encode`, by Nick Ing-Simmons, provides a mechanism to translate between different character encodings. Support for Unicode, ISO-8859-*, ASCII, CP*, KOI8-R, and three variants of EBCDIC are compiled in to the module. Several other encodings (like Japanese, Chinese, and MacIntosh encodings) are included and will be loaded at runtime.

Any encoding supported by `Encode` module is also available to the `":encoding()"` layer if `PerlIO` is used.

See [Encode](#) for more information.

- `Filter::Simple` is an easy-to-use frontend to `Filter::Util::Call`, from Damian Conway.

```
# in MyFilter.pm:
package MyFilter;
use Filter::Simple sub {
    while (my ($from, $to) = splice @_, 0, 2) {
        s/$from/$to/g;
    }
};
1;

# in user's code:
use MyFilter qr/red/ => 'green';

print "red\n"; # this code is filtered, will print "green\n"
print "bored\n"; # this code is filtered, will print "bogreen\n"
```

```
no MyFilter;

print "red\n";    # this code is not filtered, will print "red\n"
```

See [Filter::Simple](#) for more information.

- `Filter::Util::Call`, by Paul Marquess, provides you with the framework to write *Source Filters* in Perl. For most uses the frontend `Filter::Simple` is to be preferred. See [Filter::Util::Call](#) for more information.
- `Locale::Constants`, `Locale::Country`, `Locale::Currency`, and `Locale::Language`, from Neil Bowers, have been added. They provide the codes for various locale standards, such as "fr" for France, "usd" for US Dollar, and "jp" for Japanese.

```
use Locale::Country;

$country = code2country('jp');          # $country gets 'Japan'
$code    = country2code('Norway');      # $code gets 'no'
```

See [Locale::Constants](#), [Locale::Country](#), [Locale::Currency](#), and [Locale::Language](#) for more information.

- `MIME::Base64`, by Gisle Aas, allows you to encode data in base64.

```
use MIME::Base64;

$encoded = encode_base64('Aladdin:open sesame');
$decoded = decode_base64($encoded);

print $encoded, "\n"; # "QWxhZGRpbjpvYGVuIHNlc2FtZQ=="
```

See [MIME::Base64](#) for more information.

- `MIME::QuotedPrint`, by Gisle Aas, allows you to encode data in quoted-printable encoding.

```
use MIME::QuotedPrint;

$encoded = encode_qp("Smiley in Unicode: \x{263a}");
$decoded = decode_qp($encoded);

print $encoded, "\n"; # "Smiley in Unicode: =263A"
```

`MIME::QuotedPrint` has been enhanced to provide the basic methods necessary to use it with `PerlIO::Via` as in :

```
use MIME::QuotedPrint;
open($fh, ">Via (MIME::QuotedPrint)", $path)
```

See [MIME::QuotedPrint](#) for more information.

- `PerlIO::Scalar`, by Nick Ing-Simmons, provides the implementation of IO to "in memory" Perl scalars as discussed above. It also serves as an example of a loadable layer. Other future possibilities include `PerlIO::Array` and `PerlIO::Code`. See [PerlIO::Scalar](#) for more information.
- `PerlIO::Via`, by Nick Ing-Simmons, acts as a `PerlIO` layer and wraps `PerlIO` layer functionality provided by a class (typically implemented in perl code).

```
use MIME::QuotedPrint;
open($fh, ">Via (MIME::QuotedPrint)", $path)
```

This will automatically convert everything output to `$fh` to Quoted-Printable. See [PerlIO::Via](#) for more information.

- `Pod::Text::Overstrike`, by Joe Smith, has been added. It converts POD data to formatted overstrike text. See [Pod::Text::Overstrike](#) for more information.

- Switch from Damian Conway has been added. Just by saying

```
use Switch;
```

you have `switch` and `case` available in Perl.

```
use Switch;
```

```
switch ($val) {
    case 1          { print "number 1" }
    case "a"        { print "string a" }
    case [1..10,42] { print "number in list" }
    case (@array)   { print "number in list" }
    case /\w+/      { print "pattern" }
    case qr/\w+/    { print "pattern" }
    case (%hash)    { print "entry in hash" }
    case (\%hash)   { print "entry in hash" }
    case (\&sub)    { print "arg to subroutine" }
    else           { print "previous case not true" }
}
```

See [Switch](#) for more information.

- Text::Balanced from Damian Conway has been added, for extracting delimited text sequences from strings.

```
use Text::Balanced 'extract_delimited';
```

```
($a, $b) = extract_delimited("'never say never', he never said", "'", '');
```

\$a will be "'never say never'", \$b will be ', he never said'.

In addition to `extract_delimited()` there are also `extract_bracketed()`, `extract_quotelike()`, `extract_codeblock()`, `extract_variable()`, `extract_tagged()`, `extract_multiple()`, `gen_delimited_pat()`, and `gen_extract_tagged()`. With these you can implement rather advanced parsing algorithms. See [Text::Balanced](#) for more information.

- Tie::RefHash::Nestable, by Edward Avis, allows storing hash references (unlike the standard Tie::RefHash). The module is contained within Tie::RefHash.
- XS::Typemap, by Tim Jenness, is a test extension that exercises XS typemaps. Nothing gets installed but for extension writers the code is worth studying.

Updated And Improved Modules and Pragmata

- B::Deparse should be now more robust. It still far from providing a full round trip for any random piece of Perl code, though, and is under active development: expect more robustness in 5.7.2.
- Class::Struct can now define the classes in compile time.
- Math::BigFloat has undergone much fixing, and in addition the `fmod()` function now supports modulus operations.

(The fixed Math::BigFloat module is also available in CPAN for those who can't upgrade their Perl: <http://www.cpan.org/authors/id/J/JP/JPEACOCK/>)

- Devel::Peek now has an interface for the Perl memory statistics (this works only if you are using perl's malloc, and if you have compiled with debugging).
- IO::Socket has now `atmark()` method, which returns true if the socket is positioned at the out-of-band mark. The method is also exportable as a `socketatmark()` function.

- IO::Socket::INET has support for ReusePort option (if your platform supports it). The Reuse option now has an alias, ReuseAddr. For clarity you may want to prefer ReuseAddr.
- Net::Ping has been enhanced. There is now "external" protocol which uses Net::Ping::External module which runs external ping(1) and parses the output. An alpha version of Net::Ping::External is available in CPAN and in 5.7.2 the Net::Ping::External may be integrated to Perl.
- The open pragma allows layers other than ":raw" and ":crlf" when using PerlIO.
- POSIX::sigaction() is now much more flexible and robust. You can now install coderef handlers, 'DEFAULT', and 'IGNORE' handlers, installing new handlers was not atomic.
- The Test module has been significantly enhanced. Its use is greatly recommended for module writers.
- The utf8:: name space (as in the pragma) provides various Perl-callable functions to provide low level access to Perl's internal Unicode representation. At the moment only length() has been implemented.

The following modules have been upgraded from the versions at CPAN: CPAN, CGI, DB_File, File::Temp, Getopt::Long, Pod::Man, Pod::Text, Storable, Text-Tabs+Wrap.

Performance Enhancements

- Hashes now use Bob Jenkins "One-at-a-Time" hashing key algorithm (<http://burtleburtle.net/bob/hash/doobs.html>). This algorithm is reasonably fast while producing a much better spread of values than the old hashing algorithm (originally by Chris Torek, later tweaked by Ilya Zakharevich). Hash values output from the algorithm on a hash of all 3-char printable ASCII keys comes much closer to passing the DIEHARD random number generation tests. According to perlbench, this change has not affected the overall speed of Perl.
- unshift() should now be noticeably faster.

Utility Changes

- h2xs now produces template README.
- s2p has been completely rewritten in Perl. (It is in fact a full implementation of sed in Perl.)
- xsubpp now supports OUT keyword.

New Documentation

perlclib

Internal replacements for standard C library functions. (Interesting only for extension writers and Perl core hackers.)

perliol

Internals of PerlIO with layers.

README.aix

Documentation on compiling Perl on AIX has been added. AIX has several different C compilers and getting the right patch level is essential. On install README.aix will be installed as [perlaix](#).

README.bs2000

Documentation on compiling Perl on the POSIX-BC platform (an EBCDIC mainframe environment) has been added.

This was formerly known as README.posix-bc but the name was considered to be too confusing (it has nothing to do with the POSIX module or the POSIX standard). On install README.bs2000 will be installed as [perlbs2000](#).

README.macos

In perl 5.7.1 (and in the 5.6.1) the MacPerl sources have been synchronised with the standard Perl sources. To compile MacPerl some additional steps are required, and this file documents those steps. On install README.macos will be installed as [perlmacos](#).

README.mpeix

The README.mpeix has been podified, which means that this information about compiling and using Perl on the MPE/iX miniframe platform will be installed as [perlmpaix](#).

README.solaris

README.solaris has been created and Solaris wisdom from elsewhere in the Perl documentation has been collected there. On install README.solaris will be installed as [perlsolaris](#).

README.vos

The README.vos has been podified, which means that this information about compiling and using Perl on the Stratus VOS miniframe platform will be installed as [perlvos](#).

Porting/repository.pod

Documentation on how to use the Perl source repository has been added.

Installation and Configuration Improvements

- Because PerlIO is now the default on most platforms, "-perlio" doesn't get appended to the `$Config{archname}` (also known as `$^O`) anymore. Instead, if you explicitly choose not to use perlIO (Configure command line option `-Uuseperlio`), you will get "-stdio" appended.
- Another change related to the architecture name is that "-64all" (`-Duse64bitall`, or "maximally 64-bit") is appended only if your pointers are 64 bits wide. (To be exact, the `use64bitall` is ignored.)
- `APPLIB_EXP`, a less-known configuration-time definition, has been documented. It can be used to prepend site-specific directories to Perl's default search path (`@INC`), see `INSTALL` for information.
- Building Berkeley DB3 for compatibility modes for DB, NDBM, and ODBM has been documented in `INSTALL`.
- If you are on IRIX or Tru64 platforms, new profiling/debugging options have been added, see [perlhack](#) for more information about pixie and Third Degree.

New Or Improved Platforms

For the list of platforms known to support Perl, see [Supported Platforms in perlport](#).

- AIX dynamic loading should be now better supported.
- After a long pause, AmigaOS has been verified to be happy with Perl.
- EBCDIC platforms (z/OS, also known as OS/390, POSIX-BC, and VM/ESA) have been regained. Many test suite tests still fail and the co-existence of Unicode and EBCDIC isn't quite settled, but the situation is much better than with Perl 5.6. See [perlos390](#), [perlbs2000](#) (for POSIX-BC), and [perlymesa](#) for more information.
- Building perl with `-Duseithreads` or `-Duse5005threads` now works under HP-UX 10.20 (previously it only worked under 10.30 or later). You will need a thread library package installed. See `README.hpux`.
- Mac OS Classic (MacPerl has of course been available since perl 5.004 but now the source code bases of standard Perl and MacPerl have been synchronised)
- NCR MP-RAS is now supported.
- NonStop-UX is now supported.

- Amdahl UTS is now supported.
- z/OS (formerly known as OS/390, formerly known as MVS OE) has now support for dynamic loading. This is not selected by default, however, you must specify `-Dusedl` in the arguments of `Configure`.

Generic Improvements

- `Configure` no longer includes the DBM libraries (`dbm`, `gdbm`, `db`, `ndbm`) when building the Perl binary. The only exception to this is SunOS 4.x, which needs them.
- Some new `Configure` symbols, useful for extension writers:

`d_cmsg_hdr`

For struct `cmsg_hdr`.

`d_fcntl_can_lock`

Whether `fcntl()` can be used for file locking.

`d_fsync`

`d_getitimer`

`d_getpagesize`

For `getpagesize()`, though you should prefer `POSIX::sysconf(_SC_PAGE_SIZE)`

`d_msg_hdr_s`

For struct `msg_hdr`.

`need_va_copy`

Whether one needs to use `Perl_va_copy()` to copy varargs.

`d_readv`

`d_recvmsg`

`d_sendmsg`

`sig_size` The number of elements in an array needed to hold all the available signals.

`d_sockatmark`

`d_strtoq`

`d_u32align`

Whether one needs to access character data aligned by U32 sized pointers.

`d_ualarm`

`d_usleep`

- Removed `Configure` symbols: the PDP-11 memory model settings: `huge`, `large`, `medium`, `models`.
- SOCKS support is now much more robust.
- If your file system supports symbolic links you can build Perl outside of the source directory by

```
mkdir /tmp/perl/build/directory
cd /tmp/perl/build/directory
sh /path/to/perl/source/Configure -Dmksymlinks ...
```

This will create in `/tmp/perl/build/directory` a tree of symbolic links pointing to files in `/path/to/perl/source`. The original files are left unaffected. After `Configure` has finished you can just say

```
make all test
```

and Perl will be built and tested, all in `/tmp/perl/build/directory`.

Selected Bug Fixes

Numerous memory leaks and uninitialized memory accesses have been hunted down. Most importantly anonymous subs used to leak quite a bit.

- `chop(@list)` in list context returned the characters chopped in reverse order. This has been reversed to be in the right order.
- The order of DESTROYs has been made more predictable.
- `mkdir()` now ignores trailing slashes in the directory name, as mandated by POSIX.
- Attributes (like `:shared`) didn't work with `our()`.
- The PERL5OPT environment variable (for passing command line arguments to Perl) didn't work for more than a single group of options.
- The tainting behaviour of `sprintf()` has been rationalized. It does not taint the result of floating point formats anymore, making the behaviour consistent with that of string interpolation.
- All but the first argument of the IO `syswrite()` method are now optional.
- `Tie::ARRAY SPLICE` method was broken.
- `vec()` now tries to work with characters ≤ 255 when possible, but it leaves higher character values in place. In that case, if `vec()` was used to modify the string, it is no longer considered to be utf8-encoded.

Platform Specific Changes and Fixes

- Linux previously had problems related to `sockaddrlen` when using `accept()`, `recvfrom()` (in Perl: `recv()`), `getpeername()`, and `getsockname()`.
- Previously DYNIX/ptx had problems in its Configure probe for non-blocking I/O.
- Windows
 - Borland C++ v5.5 is now a supported compiler that can build Perl. However, the generated binaries continue to be incompatible with those generated by the other supported compilers (GCC and Visual C++).
 - `Win32::GetCwd()` correctly returns `C:\` instead of `C:` when at the drive root. Other bugs in `chdir()` and `Cwd::cwd()` have also been fixed.
 - Duping socket handles with `open(F, "&MYSOCK")` now works under Windows 9x.
 - HTML files will be installed in `c:\perl\html` instead of `c:\perl\lib\pod\html`
 - The makefiles now provide a single switch to bulk-enable all the features enabled in ActiveState ActivePerl (a popular binary distribution).

New or Changed Diagnostics

Two new debugging options have been added: if you have compiled your Perl with debugging, you can use the `-DT` and `-DR` options to trace tokenising and to add reference counts to displaying variables, respectively.

- If an attempt to use a (non-blessed) reference as an array index is made, a warning is given.
- `push @a;` and `unshift @a;` (with no values to push or unshift) now give a warning. This may be a problem for generated and eval'd code.

Changed Internals

- Some new APIs: `ptr_table_clear()`, `ptr_table_free()`, `sv_setref_uv()`. For the full list of the available APIs see [perlapi](#).

- dTHR and djSP have been obsoleted; the former removed (because it's a no-op) and the latter replaced with dSP.
- Perl now uses system malloc instead of Perl malloc on all 64-bit platforms, and even in some not-always-64-bit platforms like AIX, IRIX, and Solaris. This change breaks backward compatibility but Perl's malloc has problems with large address spaces and also the speed of vendors' malloc is generally better in large address space machines (Perl's malloc is mostly tuned for space).

New Tests

Many new tests have been added. The most notable is probably the lib/1_compile: it is very notable because running it takes quite a long time — it test compiles all the Perl modules in the distribution. Please be patient.

Known Problems

Note that unlike other sections in this document (which describe changes since 5.7.0) this section is cumulative containing known problems for all the 5.7 releases.

AIX vac 5.0.0.0 May Produce Buggy Code For Perl

The AIX C compiler vac version 5.0.0.0 may produce buggy code, resulting in few random tests failing, but when the failing tests are run by hand, they succeed. We suggest upgrading to at least vac version 5.0.1.0, that has been known to compile Perl correctly. "lspp -L|grep vac.C" will tell you the vac version.

lib/ftmp-security tests warn 'system possibly insecure'

Don't panic. Read INSTALL 'make test' section instead.

lib/io_multihomed Fails In LP64-Configured HP-UX

The lib/io_multihomed test may hang in HP-UX if Perl has been configured to be 64-bit. Because other 64-bit platforms do not hang in this test, HP-UX is suspect. All other tests pass in 64-bit HP-UX. The test attempts to create and connect to "multihomed" sockets (sockets which have multiple IP addresses).

Test lib/posix Subtest 9 Fails In LP64-Configured HP-UX

If perl is configured with -Duse64bitall, the successful result of the subtest 10 of lib/posix may arrive before the successful result of the subtest 9, which confuses the test harness so much that it thinks the subtest 9 failed.

lib/b test 19

The test fails on various platforms (PA64 and IA64 are known), but the exact cause is still being investigated.

Linux With Sflo Fails op/misc Test 48

No known fix.

sigaction test 13 in VMS

The test is known to fail; whether it's because of VMS or because of faulty test is not known.

sprintf tests 129 and 130

The op/sprintf tests 129 and 130 are known to fail on some platforms. Examples include any platform using sfio, and Compaq/Tandem's NonStop-UX. The failing platforms do not comply with the ANSI C Standard, line 19ff on page 134 of ANSI X3.159 1989 to be exact. (They produce something else than "1" and "-1" when formatting 0.6 and -0.6 using the printf format "%.0f", most often they produce "0" and "-0".)

Failure of Thread tests

The subtests 19 and 20 of lib/thr5005.t test are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures—Perl 5.005_0x has the same bugs, but didn't have these tests. (Note that support for 5.005-style threading remains experimental.)

Localising a Tied Variable Leaks Memory

```
use Tie::Hash;
tie my %tie_hash => 'Tie::StdHash';

...

local($tie_hash{Foo}) = 1; # leaks
```

Code like the above is known to leak memory every time the `local()` is executed.

Self-tying of Arrays and Hashes Is Forbidden

Self-tying of arrays and hashes is broken in rather deep and hard-to-fix ways. As a stop-gap measure to avoid people from getting frustrated at the mysterious results (core dumps, most often) it is for now forbidden (you will get a fatal error even from an attempt).

Building Extensions Can Fail Because Of Largefiles

Some extensions like `mod_perl` are known to have issues with ‘largefiles’, a change brought by Perl 5.6.0 in which file offsets default to 64 bits wide, where supported. Modules may fail to compile at all or compile and work incorrectly. Currently there is no good solution for the problem, but `Configure` now provides appropriate non-largefile `ccflags`, `ldflags`, `libswanted`, and `libs` in the `%Config` hash (e.g., `$Config{ccflags_nolargefiles}`) so the extensions that are having problems can try configuring themselves without the largefile-ness. This is admittedly not a clean solution, and the solution may not even work at all. One potential failure is whether one can (or, if one can, whether it’s a good idea) link together at all binaries with different ideas about file offsets, all this is platform-dependent.

The Compiler Suite Is Still Experimental

The compiler suite is slowly getting better but is nowhere near working order yet.

Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the `comp.lang.perl.misc` newsgroup and the perl bug database at <http://bugs.perl.org/>. There may also be information at <http://www.perl.com/perl/>, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to `perlbug@perl.org` to be analysed by the Perl porting team.

SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

HISTORY

Written by Jarkko Hietaniemi <jhi@iki.fi>, with many contributions from The Perl Porters and Perl Users submitting feedback and patches.

Send omissions or corrections to <perlbug@perl.org>.

NAME

perl572delta – what's new for perl v5.7.2

DESCRIPTION

This document describes differences between the 5.7.1 release and the 5.7.2 release.

(To view the differences between the 5.6.0 release and the 5.7.0 release, see [perl570delta](#). To view the differences between the 5.7.0 release and the 5.7.1 release, see [perl571delta](#).)

Security Vulnerability Closed

(This change was already made in 5.7.0 but bears repeating here.)

A security vulnerability affecting all Perl versions prior to 5.6.1 was found in August 2000. The vulnerability does not affect default installations and as far as is known affects only the Linux platform.

You should upgrade your Perl to 5.6.1 as soon as possible. Patches for earlier releases exist but using the patches require full recompilation from the source code anyway, so 5.6.1 is your best choice.

See <http://www.cpan.org/src/5.0/sperl-2000-08-05/sperl-2000-08-05.txt> for more information.

Incompatible Changes**64-bit platforms and malloc**

If your pointers are 64 bits wide, the Perl malloc is no more being used because it simply does not work with 8-byte pointers. Also, usually the system malloc on such platforms are much better optimized for such large memory models than the Perl malloc.

AIX Dynaloading

The AIX dynaloading now uses in AIX releases 4.3 and newer the native dlopen interface of AIX instead of the old emulated interface. This change will probably break backward compatibility with compiled modules.

The change was made to make Perl more compliant with other applications like modperl which are using the AIX native interface.

Socket Extension Dynamic in VMS

The Socket extension is now dynamically loaded instead of being statically built in. This may or may not be a problem with ancient TCP/IP stacks of VMS: we do not know since we weren't able to test Perl in such configurations.

Different Definition of the Unicode Character Classes \p{In...}

As suggested by the Unicode consortium, the Unicode character classes now prefer *scripts* as opposed to *blocks* (as defined by Unicode); in Perl, when the `\p{In...}` and the `\p{In...}` regular expression constructs are used. This has changed the definition of some of those character classes.

The difference between scripts and blocks is that scripts are the glyphs used by a language or a group of languages, while the blocks are more artificial groupings of 256 characters based on the Unicode numbering.

In general this change results in more inclusive Unicode character classes, but changes to the other direction also do take place: for example while the script `Latin` includes all the Latin characters and their various diacritic-adorned versions, it does not include the various punctuation or digits (since they are not solely Latin).

Changes in the character class semantics may have happened if a script and a block happen to have the same name, for example Hebrew. In such cases the script wins and `\p{InHebrew}` now means the script definition of Hebrew. The block definition is still available, though, by appending `Block` to the name: `\p{InHebrewBlock}` means what `\p{InHebrew}` meant in perl 5.6.0. For the full list of affected character classes, see [Blocks](#).

Deprecations

The current user-visible implementation of pseudo-hashes (the weird use of the first array element) is deprecated starting from Perl 5.8.0 and will be removed in Perl 5.10.0, and the feature will be implemented differently. Not only is the current interface rather ugly, but the current implementation slows down normal array and hash use quite noticeably. The `fields` pragma interface will remain available.

The syntaxes `@a-[...]` and `@h-{...}` have now been deprecated.

The `suidperl` is also considered to be too much a risk to continue maintaining and the `suidperl` code is likely to be removed in a future release.

The `package;` syntax (`package` without an argument has been deprecated. Its semantics were never that clear and its implementation even less so. If you have used that feature to disallow all but fully qualified variables, use `strict;` instead.

The `chdir(undef)` and `chdir('')` behaviors to match `chdir()` has been deprecated. In future versions, `chdir(undef)` and `chdir('')` will simply fail.

Core Enhancements

In general a lot of fixing has happened in the area of Perl's understanding of numbers, both integer and floating point. Since in many systems the standard number parsing functions like `strtoul()` and `atof()` seem to have bugs, Perl tries to work around their deficiencies. This results hopefully in more accurate numbers.

- The rules for allowing underscores (underbars) in numeric constants have been relaxed and simplified: now you can have an underscore **between digits**.
- GMAGIC (right-hand side magic) could in many cases such as string concatenation be invoked too many times.
- Lexicals I: lexicals outside an `eval ""` weren't resolved correctly inside a subroutine definition inside the `eval ""` if they were not already referenced in the top level of the `eval""`ed code.
- Lexicals II: lexicals leaked at file scope into subroutines that were declared before the lexicals.
- Lvalue subroutines can now return `undef` in list context.
- The `op_clear` and `op_null` are now exported.
- A new special regular expression variable has been introduced: `$_N`, which contains the most-recently closed group (submatch).
- [utime](#) now supports `utime undef, undef, @files` to change the file timestamps to the current time.
- The Perl parser has been stress tested using both random input and Markov chain input.
- `eval "v200"` now works.
- VMS now works under `PerlIO`.
- END blocks are now run even if you `exit/die` in a BEGIN block. The execution of END blocks is now controlled by `PL_exit_flags` & `PERL_EXIT_DESTRUCT_END`. This enables the new behaviour for perl embedders. This will default in 5.10. See [perlembed](#).

Modules and Pragmata

New Modules and Distributions

- [Attribute::Handlers](#) – Simpler definition of attribute handlers
- [ExtUtils::Constant](#) – generate XS code to import C header constants

- *I18N::Langinfo* – query locale information
- *I18N::LangTags* – functions for dealing with RFC3066–style language tags
- *libnet* – a collection of perl5 modules related to network programming
Perl installation leaves libnet unconfigured, use *libnetcfg* to configure.
- *List::Util* – selection of general–utility list subroutines
- *Locale::Maketext* – framework for localization
- *Memoize* – Make your functions faster by trading space for time
- *NEXT* – pseudo–class for method redispach
- *Scalar::Util* – selection of general–utility scalar subroutines
- *Test::More* – yet another framework for writing test scripts
- *Test::Simple* – Basic utilities for writing tests
- *Time::HiRes* – high resolution ualarm, usleep, and gettimeofday
- *Time::Piece* – Object Oriented time objects
(Previously known as *Time::Object*.)
- *Time::Seconds* – a simple API to convert seconds to other date values
- *UnicodeCD* – Unicode Character Database

Updated And Improved Modules and Pragmata

- *B::Deparse* module has been significantly enhanced. It now can deparse almost all of the standard test suite (so that the tests still succeed). There is a make target "test.deparse" for trying this out.
- *Class::Struct* now assigns the array/hash element if the accessor is called with an array/hash element as the **sole** argument.
- *Cwd* extension is now (even) faster.
- *DB_File* extension has been updated to version 1.77.
- *Fcntl*, *Socket*, and *Sys::Syslog* have been rewritten to use the new–style constant dispatch section (see *ExtUtils::Constant*).
- *File::Find* is now (again) reentrant. It also has been made more portable.
- *File::Glob* now supports GLOB_LIMIT constant to limit the size of the returned list of filenames.
- *IO::Socket::INET* now supports LocalPort of zero (usually meaning that the operating system will make one up.)
- The *vars* pragma now supports declaring fully qualified variables. (Something that *our()* does not and will not support.)

Utility Changes

- The *emacs/e2ctags.pl* is now much faster.
- *h2ph* now supports C trigraphs.
- *h2xs* uses the new *ExtUtils::Constant* module which will affect newly created extensions that define constants. Since the new code is more correct (if you have two constants where the first one is a prefix of the second one, the first constant **never** gets defined), less lossy (it uses integers for integer constant, as opposed to the old code that used floating point numbers even for integer constants), and slightly faster, you might want to consider regenerating your extension code (the new scheme makes

regenerating easy). [h2xs](#) now also supports C trigraphs.

- [libnetcfg](#) has been added to configure the libnet.
- The [Pod::Html](#) (and thusly [pod2html](#)) now allows specifying a cache directory.

New Documentation

- [Locale::Maketext::TPJ13](#) is an article about software localization, originally published in The Perl Journal #13, republished here with kind permission.
- More README.\$PLATFORM files have been converted into pod, which also means that they also be installed as perl\$PLATFORM documentation files. The new files are [perlapollo](#), [perlbeos](#), [perldgux](#), [perlhurd](#), [perlmini](#), [perlntware](#), [perlplan9](#), [perlqnx](#), and [perltru64](#).
- The [Todo](#) and [Todo-5.6](#) files have been merged into [perltodo](#).
- Use of the [gprof](#) tool to profile Perl has been documented in [perlhack](#). There is a make target "perl.gprof" for generating a gprofiled Perl executable.

Installation and Configuration Improvements

New Or Improved Platforms

- AIX should now work better with gcc, threads, and 64-bitness. Also the long doubles support in AIX should be better now. See [perlaix](#).
- AtheOS (<http://www.atheos.cx/>) is a new platform.
- DG/UX platform now supports the 5.005-style threads. See [perldgux](#).
- DYNIX/ptx platform (a.k.a. dynixptx) is supported at or near osvers 4.5.2.
- Several Mac OS (Classic) portability patches have been applied. We hope to get a fully working port by 5.8.0. (The remaining problems relate to the changed IO model of Perl.) See [perlmacos](#).
- Mac OS X (or Darwin) should now be able to build Perl even on HFS+ filesystems. (The case-insensitivity confused the Perl build process.)
- NetWare from Novell is now supported. See [perlntware](#).
- The Amdahl UTS UNIX mainframe platform is now supported.

Generic Improvements

- In AFS installations one can configure the root of the AFS to be somewhere else than the default [/afs](#) by using the Configure parameter `-Dafsroot=/some/where/else`.
- The version of Berkeley DB used when the Perl (and, presumably, the DB_File extension) was built is now available as `@Config{qw(db_version_major db_version_minor db_version_patch)}` from Perl and as `DB_VERSION_MAJOR_CFG DB_VERSION_MINOR_CFG DB_VERSION_PATCH_CFG` from C.
- The Thread extension is now not built at all under ithreads (Configure `-Duseithreads`) because it wouldn't work anyway (the Thread extension requires being Configured with `-Duse5005threads`).
- The `B::Deparse` compiler backend has been so significantly improved that almost the whole Perl test suite passes after being deparsed. A make target has been added to help in further testing: `make test.deparse`.

Selected Bug Fixes

- The `autouse` pragma didn't work for `Multi::Part::Function::Names`.

- The behaviour of non-decimal but numeric string constants such as "0x23" was platform-dependent: in some platforms that was seen as 35, in some as 0, in some as a floating point number (don't ask). This was caused by Perl using the operating system libraries in a situation where the result of the string to number conversion is undefined: now Perl consistently handles such strings as zero in numeric contexts.
- [*dproffpp*](#) -R didn't work.
- PERL5OPT with embedded spaces didn't work.
- [*Sys::Syslog*](#) ignored the LOG_AUTH constant.

Platform Specific Changes and Fixes

- Some versions of glibc have a broken `modfl()`. This affects builds with `-Duselongdouble`. This version of Perl detects this brokenness and has a workaround for it. The glibc release 2.2.2 is known to have fixed the `modfl()` bug.

New or Changed Diagnostics

- In the regular expression diagnostics the `<< HERE` marker introduced in 5.7.0 has been changed to be `<- HERE` since too many people found the `<<` to be too similar to here-document starters.
- If you try to [*pack*](#) a number less than 0 or larger than 255 using the "C" format you will get an optional warning. Similarly for the "c" format and a number less than -128 or more than 127.
- Certain regex modifiers such as `(?o)` make sense only if applied to the entire regex. You will an optional warning if you try to do otherwise.
- Using arrays or hashes as references (e.g. `%foo->{bar}`) has been deprecated for a while. Now you will get an optional warning.

Source Code Enhancements

MAGIC constants

The MAGIC constants (e.g. `'P'`) have been macrofied (e.g. `PERL_MAGIC_TIED`) for better source code readability and maintainability.

Better commented code

[*perly.c*](#), [*sv.c*](#), and [*sv.h*](#) have now been extensively commented.

Regex pre-/post-compilation items matched up

The regex compiler now maintains a structure that identifies nodes in the compiled bytecode with the corresponding syntactic features of the original regex expression. The information is attached to the new `offsets` member of the `struct regexp`. See [*perldebguts*](#) for more complete information.

gcc -Wall

The C code has been made much more `gcc -Wall` clean. Some warning messages still remain, though, so if you are compiling with `gcc` you will see some warnings about dubious practices. The warnings are being worked on.

New Tests

Several new tests have been added, especially for the [*lib*](#) subsection.

The tests are now reported in a different order than in earlier Perls. (This happens because the test scripts from under `t/lib` have been moved to be closer to the library/extension they are testing.)

Known Problems

Note that unlike other sections in this document (which describe changes since 5.7.0) this section is cumulative containing known problems for all the 5.7 releases.

AIX

- In AIX 4.2 Perl extensions that use C++ functions that use statics may have problems in that the statics are not getting initialized. In newer AIX releases this has been solved by linking Perl with the libC_r library, but unfortunately in AIX 4.2 the said library has an obscure bug where the various functions related to time (such as `time()` and `gettimeofday()`) return broken values, and therefore in AIX 4.2 Perl is not linked against the libC_r.
- vac 5.0.0.0 May Produce Buggy Code For Perl
The AIX C compiler vac version 5.0.0.0 may produce buggy code, resulting in few random tests failing, but when the failing tests are run by hand, they succeed. We suggest upgrading to at least vac version 5.0.1.0, that has been known to compile Perl correctly. "`ls|pp -L|grep vac.C`" will tell you the vac version.

Amiga Perl Invoking Mystery

One cannot call Perl using the `volume:` syntax, that is, `perl -v` works, but for example `bin:perl -v` doesn't. The exact reason is known but the current suspect is the *ixemul* library.

lib/ftmp-security tests warn 'system possibly insecure'

Don't panic. Read INSTALL 'make test' section instead.

Cygwin intermittent failures of lib/Memoize/t/expire_file 11 and 12

The subtests 11 and 12 sometimes fail and sometimes work.

HP-UX lib/io_multihomed Fails When LP64-Configured

The lib/io_multihomed test may hang in HP-UX if Perl has been configured to be 64-bit. Because other 64-bit platforms do not hang in this test, HP-UX is suspect. All other tests pass in 64-bit HP-UX. The test attempts to create and connect to "multihomed" sockets (sockets which have multiple IP addresses).

HP-UX lib/posix Subtest 9 Fails When LP64-Configured

If perl is configured with `-Duse64bitall`, the successful result of the subtest 10 of lib/posix may arrive before the successful result of the subtest 9, which confuses the test harness so much that it thinks the subtest 9 failed.

Linux With Sflo Fails op/misc Test 48

No known fix.

OS/390

OS/390 has rather many test failures but the situation is actually better than it was in 5.6.0, it's just that so many new modules and tests have been added.

Failed Test	Stat	Wstat	Total	Fail	Failed	List of Failed
../ext/B/Deparse.t			14	1	7.14%	14
../ext/B/Showlex.t			1	1	100.00%	1
../ext/Encode/Encode/Tcl.t			610	13	2.13%	592 594 596 598 600 602 604-610
../ext/IO/lib/IO/t/io_unix.t	113	28928	5	3	60.00%	3-5
../ext/POSIX/POSIX.t			29	1	3.45%	14
../ext/Storable/t/lock.t	255	65280	5	3	60.00%	3-5
../lib/locale.t	129	33024	117	19	16.24%	99-117
../lib/warnings.t			434	1	0.23%	75
../lib/ExtUtils.t			27	1	3.70%	25
../lib/Math/BigInt/t/bigintpm.t			1190	1	0.08%	1145
../lib/Unicode/UCD.t			81	48	59.26%	1-16 49-64 66-81
../lib/User/pwent.t			9	1	11.11%	4
op/pat.t			660	6	0.91%	242-243 424-425

```

                                626-627
op/split.t                      0      9    ??  ??    %  ??
op/taint.t                     174     3   1.72% 156 162 168
op/tr.t                         70     3   4.29%  50 58-59
Failed 16/422 test scripts, 96.21% okay. 105/23251 subtests failed, 99.55% okay.

```

op/sprintf tests 129 and 130

The op/sprintf tests 129 and 130 are known to fail on some platforms. Examples include any platform using sfio, and Compaq/Tandem's NonStop-UX. The failing platforms do not comply with the ANSI C Standard, line 19ff on page 134 of ANSI X3.159 1989 to be exact. (They produce something other than "1" and "-1" when formatting 0.6 and -0.6 using the printf format "%.0f", most often they produce "0" and "-0".)

Failure of Thread tests

Note that support for 5.005-style threading remains experimental.

The following tests are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures—Perl 5.005_0x has the same bugs, but didn't have these tests.

```

lib/autouse.t                   4
t/lib/thr5005.t                19-20

```

UNICOS

- ext/POSIX/sigaction subtests 6 and 13 may fail.
- lib/ExtUtils may spuriously claim that subtest 28 failed, which is interesting since the test only has 27 tests.
- Numerous numerical test failures

```

op/numconvert                   209,210,217,218
op/override                     7
ext/Time/HiRes/HiRes           9
lib/Math/BigInt/t/bigintpm     1145
lib/Math/Trig                   25

```

These tests fail because of yet unresolved floating point inaccuracies.

UTS

There are a few known test failures, see [perluts](#).

VMS

Rather many tests are failing in VMS but that actually more tests succeed in VMS than they used to, it's just that there are many, many more tests than there used to be.

Here are the known failures from some compiler/platform combinations.

DEC C V5.3-006 on OpenVMS VAX V6.2

```

[-.ext.list.util.t] tainted.....FAILED on test 3
[-.ext.posix] sigaction.....FAILED on test 7
[-.ext.time.hires] hires.....FAILED on test 14
[-.lib.file.find] taint.....FAILED on test 17
[-.lib.math.bigint.t] bigintpm.....FAILED on test 1183
[-.lib.test.simple.t] exit.....FAILED on test 1
[.lib] vmsish.....FAILED on test 13
[.op] sprintf.....FAILED on test 12
Failed 8/399 tests, 91.23% okay.

```

DEC C V6.0-001 on OpenVMS Alpha V7.2-1 and Compaq C V6.2-008 on OpenVMS Alpha V7.1

```

[-.ext.list.util.t] tainted.....FAILED on test 3

```



```
[-.lib.file.find]taint.....FAILED on test 17
[-.lib.test.simple.t]exit.....FAILED on test 1
[.lib]vmsish.....FAILED on test 13
Failed 4/399 tests, 92.48% okay.
```

Compaq C V6.4-005 on OpenVMS Alpha 7.2.1

```
[-.ext.b]showlex.....FAILED on test 1
[-.ext.list.util.t]tainted.....FAILED on test 3
[-.lib.file.find]taint.....FAILED on test 17
[-.lib.test.simple.t]exit.....FAILED on test 1
[.lib]vmsish.....FAILED on test 13
[.op]misc.....FAILED on test 49
Failed 6/401 tests, 92.77% okay.
```

Win32

In multi-CPU boxes there are some problems with the I/O buffering: some output may appear twice.

Localising a Tied Variable Leaks Memory

```
use Tie::Hash;
tie my %tie_hash => 'Tie::StdHash';

...

local($tie_hash{Foo}) = 1; # leaks
```

Code like the above is known to leak memory every time the `local()` is executed.

Self-tying of Arrays and Hashes Is Forbidden

Self-tying of arrays and hashes is broken in rather deep and hard-to-fix ways. As a stop-gap measure to avoid people from getting frustrated at the mysterious results (core dumps, most often) it is for now forbidden (you will get a fatal error even from an attempt).

Variable Attributes are not Currently Usable for Tying

This limitation will hopefully be fixed in future. (Subroutine attributes work fine for tying, see [Attribute::Handlers](#)).

Building Extensions Can Fail Because Of Largefiles

Some extensions like `mod_perl` are known to have issues with 'largefiles', a change brought by Perl 5.6.0 in which file offsets default to 64 bits wide, where supported. Modules may fail to compile at all or compile and work incorrectly. Currently there is no good solution for the problem, but `Configure` now provides appropriate non-largefile `ccflags`, `ldflags`, `libswanted`, and `libs` in the `%Config` hash (e.g., `$Config{ccflags_nolargefiles}`) so the extensions that are having problems can try configuring themselves without the largefile-ness. This is admittedly not a clean solution, and the solution may not even work at all. One potential failure is whether one can (or, if one can, whether it's a good idea) link together at all binaries with different ideas about file offsets, all this is platform-dependent.

The Compiler Suite Is Still Experimental

The compiler suite is slowly getting better but is nowhere near working order yet.

The Long Double Support is Still Experimental

The ability to configure Perl's numbers to use "long doubles", floating point numbers of hopefully better accuracy, is still experimental. The implementations of long doubles are not yet widespread and the existing implementations are not quite mature or standardised, therefore trying to support them is a rare and moving target. The gain of more precision may also be offset by slowdown in computations (more bits to move around, and the operations are more likely to be executed by less optimised libraries).

Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the comp.lang.perl.misc newsgroup and the perl bug database at <http://bugs.perl.org/>. There may also be information at <http://www.perl.com/perl/>, the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to `perlbug@perl.org` to be analysed by the Perl porting team.

SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

HISTORY

Written by Jarkko Hietaniemi <jhi@iki.fi>, with many contributions from The Perl Porters and Perl Users submitting feedback and patches.

Send omissions or corrections to <perlbug@perl.org>.

NAME

perlapi – autogenerated documentation for the perl public API

DESCRIPTION

This file contains the documentation of the perl public API generated by `embed.pl`, specifically a listing of functions, macros, flags, and variables that may be used by extension writers. The interfaces of any functions that are not listed here are subject to change without notice. For this reason, blindly using functions listed in `proto.h` is to be avoided when writing extensions.

Note that all Perl API global variables must be referenced with the `PL_` prefix. Some macros are provided for compatibility with the older, unadorned names, but this support may be disabled in a future release.

The listing is alphabetical, case insensitive.

"Gimme" Values

GIMME A backward-compatible version of `GIMME_V` which can only return `G_SCALAR` or `G_ARRAY`; in a void context, it returns `G_SCALAR`. Deprecated. Use `GIMME_V` instead.

U32 GIMME

=for hackers Found in file `op.h`

GIMME_V

The XSUB-writer's equivalent to Perl's `wantarray`. Returns `G_VOID`, `G_SCALAR` or `G_ARRAY` for void, scalar or list context, respectively.

U32 GIMME_V

=for hackers Found in file `op.h`

G_ARRAY

Used to indicate list context. See `GIMME_V`, `GIMME` and [perlcall](#).

=for hackers Found in file `cop.h`

G_DISCARD

Indicates that arguments returned from a callback should be discarded. See [perlcall](#).

=for hackers Found in file `cop.h`

G_EVAL Used to force a Perl `eval` wrapper around a callback. See [perlcall](#).

=for hackers Found in file `cop.h`

G_NOARGS

Indicates that no arguments are being sent to a callback. See [perlcall](#).

=for hackers Found in file `cop.h`

G_SCALAR

Used to indicate scalar context. See `GIMME_V`, `GIMME`, and [perlcall](#).

=for hackers Found in file `cop.h`

G_VOID Used to indicate void context. See `GIMME_V` and [perlcall](#).

=for hackers Found in file `cop.h`

Array Manipulation Functions

AvFILL Same as `av_len()`. Deprecated, use `av_len()` instead.

int AvFILL(AV* av)

=for hackers Found in file av.h

av_clear Clears an array, making it empty. Does not free the memory used by the array itself.

```
void    av_clear(AV* ar)
```

=for hackers Found in file av.c

av_delete

Deletes the element indexed by *key* from the array. Returns the deleted element. *flags* is currently ignored.

```
SV*     av_delete(AV* ar, I32 key, I32 flags)
```

=for hackers Found in file av.c

av_exists Returns true if the element indexed by *key* has been initialized.

This relies on the fact that uninitialized array elements are set to `&PL_sv_undef`.

```
bool     av_exists(AV* ar, I32 key)
```

=for hackers Found in file av.c

av_extend

Pre-extend an array. The *key* is the index to which the array should be extended.

```
void     av_extend(AV* ar, I32 key)
```

=for hackers Found in file av.c

av_fetch Returns the SV at the specified index in the array. The *key* is the index. If *lval* is set then the fetch will be part of a store. Check that the return value is non-null before dereferencing it to a SV*.

See [Understanding the Magic of Tied Hashes and Arrays in perlguits](#) for more information on how to use this function on tied arrays.

```
SV**     av_fetch(AV* ar, I32 key, I32 lval)
```

=for hackers Found in file av.c

av_fill Ensure that an array has a given number of elements, equivalent to Perl's `$#array = $fill;`.

```
void     av_fill(AV* ar, I32 fill)
```

=for hackers Found in file av.c

av_len Returns the highest index in the array. Returns -1 if the array is empty.

```
I32      av_len(AV* ar)
```

=for hackers Found in file av.c

av_make Creates a new AV and populates it with a list of SVs. The SVs are copied into the array, so they may be freed after the call to `av_make`. The new AV will have a reference count of 1.

```
AV*      av_make(I32 size, SV** svp)
```

=for hackers Found in file av.c

av_pop Pops an SV off the end of the array. Returns `&PL_sv_undef` if the array is empty.

```
SV*      av_pop(AV* ar)
```

=for hackers Found in file av.c

av_push Pushes an SV onto the end of the array. The array will grow automatically to accommodate the addition.

```
void av_push(AV* ar, SV* val)
```

=for hackers Found in file av.c

av_shift Shifts an SV off the beginning of the array.

```
SV* av_shift(AV* ar)
```

=for hackers Found in file av.c

av_store Stores an SV in an array. The array index is specified as key. The return value will be NULL if the operation failed or if the value did not need to be actually stored within the array (as in the case of tied arrays). Otherwise it can be dereferenced to get the original SV*. Note that the caller is responsible for suitably incrementing the reference count of val before the call, and decrementing it if the function returned NULL.

See [Understanding the Magic of Tied Hashes and Arrays in perlguits](#) for more information on how to use this function on tied arrays.

```
SV** av_store(AV* ar, I32 key, SV* val)
```

=for hackers Found in file av.c

av_undef Undefines the array. Frees the memory used by the array itself.

```
void av_undef(AV* ar)
```

=for hackers Found in file av.c

av_unshift

Unshift the given number of undef values onto the beginning of the array. The array will grow automatically to accommodate the addition. You must then use av_store to assign values to these new elements.

```
void av_unshift(AV* ar, I32 num)
```

=for hackers Found in file av.c

get_av Returns the AV of the specified Perl array. If create is set and the Perl variable does not exist then it will be created. If create is not set and the variable does not exist then NULL is returned.

NOTE: the perl_ form of this function is deprecated.

```
AV* get_av(const char* name, I32 create)
```

=for hackers Found in file perl.c

newAV Creates a new AV. The reference count is set to 1.

```
AV* newAV()
```

=for hackers Found in file av.c

Nullav Null AV pointer.

=for hackers Found in file av.h

sortsv Sort an array. Here is an example:

```
sortsv(AvARRAY(av), av_len(av)+1, Perl_sv_cmp_locale);
```

See lib/sort.pm for details about controlling the sorting algorithm.

```
void      sortsv(SV ** array, size_t num_elts, SVCOMPARE_t cmp)
=for hackers Found in file pp_sort.c
```

Callback Functions

call_argv Performs a callback to the specified Perl sub. See [perlcall](#).

NOTE: the perl_ form of this function is deprecated.

```
I32      call_argv(const char* sub_name, I32 flags, char** argv)
=for hackers Found in file perl.c
```

call_method

Performs a callback to the specified Perl method. The blessed object must be on the stack. See [perlcall](#).

NOTE: the perl_ form of this function is deprecated.

```
I32      call_method(const char* methname, I32 flags)
=for hackers Found in file perl.c
```

call_pv Performs a callback to the specified Perl sub. See [perlcall](#).

NOTE: the perl_ form of this function is deprecated.

```
I32      call_pv(const char* sub_name, I32 flags)
=for hackers Found in file perl.c
```

call_sv Performs a callback to the Perl sub whose name is in the SV. See [perlcall](#).

NOTE: the perl_ form of this function is deprecated.

```
I32      call_sv(SV* sv, I32 flags)
=for hackers Found in file perl.c
```

ENTER Opening bracket on a callback. See LEAVE and [perlcall](#).

```
ENTER;
=for hackers Found in file scope.h
```

eval_pv Tells Perl to eval the given string and return an SV* result.

NOTE: the perl_ form of this function is deprecated.

```
SV*      eval_pv(const char* p, I32 croak_on_error)
=for hackers Found in file perl.c
```

eval_sv Tells Perl to eval the string in the SV.

NOTE: the perl_ form of this function is deprecated.

```
I32      eval_sv(SV* sv, I32 flags)
=for hackers Found in file perl.c
```

FREETMPS

Closing bracket for temporaries on a callback. See SAVETMPS and [perlcall](#).

```
FREETMPS;
=for hackers Found in file scope.h
```

LEAVE Closing bracket on a callback. See ENTER and [perlcalls](#).

`LEAVE;`

=for hackers Found in file scope.h

SAVETMPS

Opening bracket for temporaries on a callback. See FREETMPS and [perlcalls](#).

`SAVETMPS;`

=for hackers Found in file scope.h

Character classes

isALNUM Returns a boolean indicating whether the C char is an ASCII alphanumeric character (including underscore) or digit.

`bool isALNUM(char ch)`

=for hackers Found in file handy.h

isALPHA Returns a boolean indicating whether the C char is an ASCII alphabetic character.

`bool isALPHA(char ch)`

=for hackers Found in file handy.h

isDIGIT Returns a boolean indicating whether the C char is an ASCII digit.

`bool isDIGIT(char ch)`

=for hackers Found in file handy.h

isLOWER

Returns a boolean indicating whether the C char is a lowercase character.

`bool isLOWER(char ch)`

=for hackers Found in file handy.h

isSPACE Returns a boolean indicating whether the C char is whitespace.

`bool isSPACE(char ch)`

=for hackers Found in file handy.h

isUPPER Returns a boolean indicating whether the C char is an uppercase character.

`bool isUPPER(char ch)`

=for hackers Found in file handy.h

toLOWER

Converts the specified character to lowercase.

`char toLOWER(char ch)`

=for hackers Found in file handy.h

toUPPER Converts the specified character to uppercase.

`char toUPPER(char ch)`

=for hackers Found in file handy.h

Cloning an interpreter**perl_clone**

Create and return a new interpreter by cloning the current one.

```
PerlInterpreter* perl_clone(PerlInterpreter* interp, UV flags)
```

=for hackers Found in file sv.c

CV Manipulation Functions**CvSTASH**

Returns the stash of the CV.

```
HV* CvSTASH(CV* cv)
```

=for hackers Found in file cv.h

get_cvReturns the CV of the specified Perl subroutine. If `create` is set and the Perl subroutine does not exist then it will be declared (which has the same effect as saying `sub name;`). If `create` is not set and the subroutine does not exist then `NULL` is returned.NOTE: the `perl_` form of this function is deprecated.

```
CV* get_cv(const char* name, I32 create)
```

=for hackers Found in file perl.c

Nullcv

Null CV pointer.

=for hackers Found in file cv.h

Embedding Functions**load_module**

Loads the module whose name is pointed to by the string part of `name`. Note that the actual module name, not its filename, should be given. Eg, "Foo::Bar" instead of "Foo/Bar.pm". `flags` can be any of `PERL_LOADMOD_DENY`, `PERL_LOADMOD_NOIMPORT`, or `PERL_LOADMOD_IMPORT_OPS` (or 0 for no flags). `ver`, if specified, provides version semantics similar to use `Foo::Bar VERSION`. The optional trailing `SV*` arguments can be used to specify arguments to the module's `import()` method, similar to use `Foo::Bar VERSION LIST`.

```
void load_module(U32 flags, SV* name, SV* ver, ...)
```

=for hackers Found in file op.c

nothreadhookStub that provides thread hook for `perl_destruct` when there are no threads.

```
int nothreadhook()
```

=for hackers Found in file perl.c

perl_alloc Allocates a new Perl interpreter. See [perlembed](#).

```
PerlInterpreter* perl_alloc()
```

=for hackers Found in file perl.c

perl_constructInitializes a new Perl interpreter. See [perlembed](#).

```
void perl_construct(PerlInterpreter* interp)
```

=for hackers Found in file perl.c

perl_destruct

Shuts down a Perl interpreter. See [perlembed](#).

```
int      perl_destruct(PerlInterpreter* interp)
```

=for hackers Found in file perl.c

perl_free Releases a Perl interpreter. See [perlembed](#).

```
void     perl_free(PerlInterpreter* interp)
```

=for hackers Found in file perl.c

perl_parse

Tells a Perl interpreter to parse a Perl script. See [perlembed](#).

```
int      perl_parse(PerlInterpreter* interp, XSINIT_t xsinit, int argc,
```

=for hackers Found in file perl.c

perl_run Tells a Perl interpreter to run. See [perlembed](#).

```
int      perl_run(PerlInterpreter* interp)
```

=for hackers Found in file perl.c

require_pv

Tells Perl to require the file named by the string argument. It is analogous to the Perl code `eval "require '$file'";`. It's even implemented that way; consider using `Perl_load_module` instead.

NOTE: the `perl_` form of this function is deprecated.

```
void     require_pv(const char* pv)
```

=for hackers Found in file perl.c

Functions in file pp_pack.c

pack_cat The engine implementing `pack()` Perl function.

```
void     pack_cat(SV *cat, char *pat, char *patend, SV **beglist, SV *
```

=for hackers Found in file pp_pack.c

unpack_str

The engine implementing `unpack()` Perl function.

```
I32      unpack_str(char *pat, char *patend, char *s, char *strbeg, ch
```

=for hackers Found in file pp_pack.c

Global Variables**PL_modglobal**

`PL_modglobal` is a general purpose, interpreter global HV for use by extensions that need to keep information on a per-interpreter basis. In a pinch, it can also be used as a symbol table for extensions to share data among each other. It is a good idea to use keys prefixed by the package name of the extension that owns the data.

```
HV*      PL_modglobal
```

=for hackers Found in file intrpvar.h

PL_na A convenience variable which is typically used with `SvPV` when one doesn't care about the length of the string. It is usually more efficient to either declare a local variable and use that instead or to use the `SvPV_nolen` macro.

```
STRLEN PL_na
```

=for hackers Found in file `thrdvar.h`

PL_sv_no

This is the `false` SV. See `PL_sv_yes`. Always refer to this as `&PL_sv_no`.

```
SV PL_sv_no
```

=for hackers Found in file `intrpvar.h`

PL_sv_undef

This is the `undef` SV. Always refer to this as `&PL_sv_undef`.

```
SV PL_sv_undef
```

=for hackers Found in file `intrpvar.h`

PL_sv_yes

This is the `true` SV. See `PL_sv_no`. Always refer to this as `&PL_sv_yes`.

```
SV PL_sv_yes
```

=for hackers Found in file `intrpvar.h`

GV Functions

GvSV Return the SV from the GV.

```
SV* GvSV(GV* gv)
```

=for hackers Found in file `gv.h`

gv_fetchmeth

Returns the glob with the given name and a defined subroutine or `NULL`. The glob lives in the given `stash`, or in the stashes accessible via `@ISA` and `UNIVERSAL::`.

The argument `level` should be either 0 or -1. If `level==0`, as a side-effect creates a glob with the given name in the given `stash` which in the case of success contains an alias for the subroutine, and sets up caching info for this glob. Similarly for all the searched stashes.

This function grants "SUPER" token as a postfix of the stash name. The GV returned from `gv_fetchmeth` may be a method cache entry, which is not visible to Perl code. So when calling `call_sv`, you should not use the GV directly; instead, you should use the method's CV, which can be obtained from the GV with the `GvCV` macro.

```
GV* gv_fetchmeth(HV* stash, const char* name, STRLEN len, I32 level)
```

=for hackers Found in file `gv.c`

gv_fetchmethod

See [gv_fetchmethod_autoload](#).

```
GV* gv_fetchmethod(HV* stash, const char* name)
```

=for hackers Found in file `gv.c`

gv_fetchmethod_autoload

Returns the glob which contains the subroutine to call to invoke the method on the `stash`. In fact in the presence of autoloading this may be the glob for "AUTOLOAD". In this case the corresponding variable `$AUTOLOAD` is already setup.

The third parameter of `gv_fetchmethod_autoload` determines whether AUTOLOAD lookup is performed if the given method is not present: non-zero means yes, look for AUTOLOAD; zero means no, don't look for AUTOLOAD. Calling `gv_fetchmethod` is equivalent to calling `gv_fetchmethod_autoload` with a non-zero `autoload` parameter.

These functions grant "SUPER" token as a prefix of the method name. Note that if you want to keep the returned glob for a long time, you need to check for it being "AUTOLOAD", since at the later time the call may load a different subroutine due to `$AUTOLOAD` changing its value. Use the glob created via a side effect to do this.

These functions have the same side-effects and as `gv_fetchmeth` with `level==0`. `name` should be writable if contains ``:'` or `'`'`. The warning against passing the GV returned by `gv_fetchmeth` to `call_sv` apply equally to these functions.

```
GV*      gv_fetchmethod_autoload(HV* stash, const char* name, I32 autoload)
```

=for hackers Found in file gv.c

gv_fetchmeth_autoload

Same as `gv_fetchmeth()`, but looks for autoloaded subroutines too. Returns a glob for the subroutine.

For an autoloaded subroutine without a GV, will create a GV even if `level < 0`. For an autoloaded subroutine without a stub, `GvCV()` of the result may be zero.

```
GV*      gv_fetchmeth_autoload(HV* stash, const char* name, STRLEN len)
```

=for hackers Found in file gv.c

gv_stashpv

Returns a pointer to the stash for a specified package. `name` should be a valid UTF-8 string. If `create` is set then the package will be created if it does not already exist. If `create` is not set and the package does not exist then NULL is returned.

```
HV*      gv_stashpv(const char* name, I32 create)
```

=for hackers Found in file gv.c

gv_stashsv

Returns a pointer to the stash for a specified package, which must be a valid UTF-8 string. See `gv_stashpv`.

```
HV*      gv_stashsv(SV* sv, I32 create)
```

=for hackers Found in file gv.c

Handy Values

HEf_SVKEY

This flag, used in the length slot of hash entries and magic structures, specifies the structure contains an `SV*` pointer where a `char*` pointer is to be expected. (For information only—not to be used).

=for hackers Found in file hv.h

Nullch Null character pointer. =for hackers Found in file handy.h

Nullsv Null SV pointer.

=for hackers Found in file handy.h

Hash Manipulation Functions

get_hv Returns the HV of the specified Perl hash. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

NOTE: the `perl_` form of this function is deprecated.

```
HV*      get_hv(const char* name, I32 create)
```

=for hackers Found in file `perl.c`

HeHASH Returns the computed hash stored in the hash entry.

```
U32      HeHASH(HE* he)
```

=for hackers Found in file `hv.h`

HeKEY Returns the actual pointer stored in the key slot of the hash entry. The pointer may be either `char*` or `SV*`, depending on the value of `HeKLEN()`. Can be assigned to. The `HePV()` or `HeSVKEY()` macros are usually preferable for finding the value of a key.

```
void*    HeKEY(HE* he)
```

=for hackers Found in file `hv.h`

HeKLEN If this is negative, and amounts to `HEf_SVKEY`, it indicates the entry holds an `SV*` key. Otherwise, holds the actual length of the key. Can be assigned to. The `HePV()` macro is usually preferable for finding key lengths.

```
STRLEN   HeKLEN(HE* he)
```

=for hackers Found in file `hv.h`

HePV Returns the key slot of the hash entry as a `char*` value, doing any necessary dereferencing of possibly `SV*` keys. The length of the string is placed in `len` (this is a macro, so do *not* use `&len`). If you do not care about what the length of the key is, you may use the global variable `PL_na`, though this is rather less efficient than using a local variable. Remember though, that hash keys in perl are free to contain embedded nulls, so using `strlen()` or similar is not a good way to find the length of hash keys. This is very similar to the `SvPV()` macro described elsewhere in this document.

```
char*    HePV(HE* he, STRLEN len)
```

=for hackers Found in file `hv.h`

HeSVKEY

Returns the key as an `SV*`, or `Nullsv` if the hash entry does not contain an `SV*` key.

```
SV*      HeSVKEY(HE* he)
```

=for hackers Found in file `hv.h`

HeSVKEY_force

Returns the key as an `SV*`. Will create and return a temporary mortal `SV*` if the hash entry contains only a `char*` key.

```
SV*      HeSVKEY_force(HE* he)
```

=for hackers Found in file `hv.h`

HeSVKEY_set

Sets the key to a given `SV*`, taking care to set the appropriate flags to indicate the presence of an `SV*` key, and returns the same `SV*`.

- `SV* HeSVKEY_set (HE* he, SV* sv)`
 =for hackers Found in file hv.h
- HeVAL** Returns the value slot (type SV*) stored in the hash entry.
`SV* HeVAL (HE* he)`
 =for hackers Found in file hv.h
- HvNAME** Returns the package name of a stash. See SvSTASH, CvSTASH.
`char* HvNAME (HV* stash)`
 =for hackers Found in file hv.h
- hv_clear** Clears a hash, making it empty.
`void hv_clear (HV* tb)`
 =for hackers Found in file hv.c
- hv_delete**
 Deletes a key/value pair in the hash. The value SV is removed from the hash and returned to the caller. The `klen` is the length of the key. The `flags` value will normally be zero; if set to `G_DISCARD` then NULL will be returned.
`SV* hv_delete (HV* tb, const char* key, I32 klen, I32 flags)`
 =for hackers Found in file hv.c
- hv_delete_ent**
 Deletes a key/value pair in the hash. The value SV is removed from the hash and returned to the caller. The `flags` value will normally be zero; if set to `G_DISCARD` then NULL will be returned. `hash` can be a valid precomputed hash value, or 0 to ask for it to be computed.
`SV* hv_delete_ent (HV* tb, SV* key, I32 flags, U32 hash)`
 =for hackers Found in file hv.c
- hv_exists** Returns a boolean indicating whether the specified hash key exists. The `klen` is the length of the key.
`bool hv_exists (HV* tb, const char* key, I32 klen)`
 =for hackers Found in file hv.c
- hv_exists_ent**
 Returns a boolean indicating whether the specified hash key exists. `hash` can be a valid precomputed hash value, or 0 to ask for it to be computed.
`bool hv_exists_ent (HV* tb, SV* key, U32 hash)`
 =for hackers Found in file hv.c
- hv_fetch** Returns the SV which corresponds to the specified key in the hash. The `klen` is the length of the key. If `lval` is set then the fetch will be part of a store. Check that the return value is non-null before dereferencing it to an SV*.
 See [Understanding the Magic of Tied Hashes and Arrays in perlguits](#) for more information on how to use this function on tied hashes.
`SV** hv_fetch (HV* tb, const char* key, I32 klen, I32 lval)`
 =for hackers Found in file hv.c

hv_fetch_ent

Returns the hash entry which corresponds to the specified key in the hash. `hash` must be a valid precomputed hash number for the given key, or 0 if you want the function to compute it. If `lval` is set then the fetch will be part of a store. Make sure the return value is non-null before accessing it. The return value when `tb` is a tied hash is a pointer to a static location, so be sure to make a copy of the structure if you need to store it somewhere.

See [Understanding the Magic of Tied Hashes and Arrays in perlguits](#) for more information on how to use this function on tied hashes.

```
HE*      hv_fetch_ent(HV* tb, SV* key, I32 lval, U32 hash)
```

=for hackers Found in file hv.c

hv_iterinit Prepares a starting point to traverse a hash table. Returns the number of keys in the hash (i.e. the same as `HvKEYS(tb)`). The return value is currently only meaningful for hashes without tie magic.

NOTE: Before version 5.004_65, `hv_iterinit` used to return the number of hash buckets that happen to be in use. If you still need that esoteric value, you can get it through the macro `HvFILL(tb)`.

```
I32      hv_iterinit(HV* tb)
```

=for hackers Found in file hv.c

hv_iterkey

Returns the key from the current position of the hash iterator. See `hv_iterinit`.

```
char*    hv_iterkey(HE* entry, I32* retlen)
```

=for hackers Found in file hv.c

hv_iterkeysv

Returns the key as an `SV*` from the current position of the hash iterator. The return value will always be a mortal copy of the key. Also see `hv_iterinit`.

```
SV*      hv_iterkeysv(HE* entry)
```

=for hackers Found in file hv.c

hv_itternext

Returns entries from a hash iterator. See `hv_iterinit`.

You may call `hv_delete` or `hv_delete_ent` on the hash entry that the iterator currently points to, without losing your place or invalidating your iterator. Note that in this case the current entry is deleted from the hash with your iterator holding the last reference to it. Your iterator is flagged to free the entry on the next call to `hv_itternext`, so you must not discard your iterator immediately else the entry will leak – call `hv_itternext` to trigger the resource deallocation.

```
HE*      hv_itternext(HV* tb)
```

=for hackers Found in file hv.c

hv_itternextsv

Performs an `hv_itternext`, `hv_iterkey`, and `hv_interval` in one operation.

```
SV*      hv_itternextsv(HV* hv, char** key, I32* retlen)
```

=for hackers Found in file hv.c

hv_iternext_flags

Returns entries from a hash iterator. See `hv_iterinit` and `hv_iternext`. The flags value will normally be zero; if `HV_ITERNEXT_WANTPLACEHOLDERS` is set the placeholders keys (for restricted hashes) will be returned in addition to normal keys. By default placeholders are automatically skipped over. Currently a placeholder is implemented with a value that is literally `<&Perl_sv_undef` (a regular `undef` value is a normal read-write SV for which `!SvOK` is false). Note that the implementation of placeholders and restricted hashes may change, and the implementation currently is insufficiently abstracted for any change to be tidy.

NOTE: this function is experimental and may change or be removed without notice.

```
HE*      hv_iternext_flags(HV* tb, I32 flags)
```

=for hackers Found in file hv.c

hv_interval Returns the value from the current position of the hash iterator. See `hv_iterkey`.

```
SV*      hv_interval(HV* tb, HE* entry)
```

=for hackers Found in file hv.c

hv_magic Adds magic to a hash. See `sv_magic`.

```
void     hv_magic(HV* hv, GV* gv, int how)
```

=for hackers Found in file hv.c

hv_store Stores an SV in a hash. The hash key is specified as `key` and `klen` is the length of the key. The `hash` parameter is the precomputed hash value; if it is zero then Perl will compute it. The return value will be NULL if the operation failed or if the value did not need to be actually stored within the hash (as in the case of tied hashes). Otherwise it can be dereferenced to get the original SV*. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL.

See [Understanding the Magic of Tied Hashes and Arrays in perlguits](#) for more information on how to use this function on tied hashes.

```
SV**     hv_store(HV* tb, const char* key, I32 klen, SV* val, U32 hash)
```

=for hackers Found in file hv.c

hv_store_ent

Stores `val` in a hash. The hash key is specified as `key`. The `hash` parameter is the precomputed hash value; if it is zero then Perl will compute it. The return value is the new hash entry so created. It will be NULL if the operation failed or if the value did not need to be actually stored within the hash (as in the case of tied hashes). Otherwise the contents of the return value can be accessed using the `He?` macros described here. Note that the caller is responsible for suitably incrementing the reference count of `val` before the call, and decrementing it if the function returned NULL.

See [Understanding the Magic of Tied Hashes and Arrays in perlguits](#) for more information on how to use this function on tied hashes.

```
HE*      hv_store_ent(HV* tb, SV* key, SV* val, U32 hash)
```

=for hackers Found in file hv.c

hv_undef Undefines the hash.

```
void     hv_undef(HV* tb)
```

=for hackers Found in file hv.c

newHV Creates a new HV. The reference count is set to 1.

HV* newHV()

=for hackers Found in file hv.c

Nullhv Null HV pointer.

=for hackers Found in file hv.h

Magical Functions

mg_clear Clear something magical that the SV represents. See `sv_magic`.

int mg_clear(SV* sv)

=for hackers Found in file mg.c

mg_copy Copies the magic from one SV to another. See `sv_magic`.

int mg_copy(SV* sv, SV* nsv, const char* key, I32 klen)

=for hackers Found in file mg.c

mg_find Finds the magic pointer for type matching the SV. See `sv_magic`.

MAGIC* mg_find(SV* sv, int type)

=for hackers Found in file mg.c

mg_free Free any magic storage used by the SV. See `sv_magic`.

int mg_free(SV* sv)

=for hackers Found in file mg.c

mg_get Do magic after a value is retrieved from the SV. See `sv_magic`.

int mg_get(SV* sv)

=for hackers Found in file mg.c

mg_length

Report on the SV's length. See `sv_magic`.

U32 mg_length(SV* sv)

=for hackers Found in file mg.c

mg_magical

Turns on the magical status of an SV. See `sv_magic`.

void mg_magical(SV* sv)

=for hackers Found in file mg.c

mg_set Do magic after a value is assigned to the SV. See `sv_magic`.

int mg_set(SV* sv)

=for hackers Found in file mg.c

SvGETMAGIC

Invokes `mg_get` on an SV if it has 'get' magic. This macro evaluates its argument more than once.

void SvGETMAGIC(SV* sv)

=for hackers Found in file sv.h

SvLOCK Arranges for a mutual exclusion lock to be obtained on `sv` if a suitable module has been loaded.

```
void    SvLOCK(SV* sv)
```

=for hackers Found in file `sv.h`

SvSETMAGIC

Invokes `mg_set` on an `SV` if it has ‘set’ magic. This macro evaluates its argument more than once.

```
void    SvSETMAGIC(SV* sv)
```

=for hackers Found in file `sv.h`

SvSetMagicSV

Like `SvSetSV`, but does any set magic required afterwards.

```
void    SvSetMagicSV(SV* dsb, SV* ssv)
```

=for hackers Found in file `sv.h`

SvSetMagicSV_nosteal

Like `SvSetMagicSV`, but does any set magic required afterwards.

```
void    SvSetMagicSV_nosteal(SV* dsb, SV* ssv)
```

=for hackers Found in file `sv.h`

SvSetSV Calls `sv_setsv` if `dsb` is not the same as `ssv`. May evaluate arguments more than once.

```
void    SvSetSV(SV* dsb, SV* ssv)
```

=for hackers Found in file `sv.h`

SvSetSV_nosteal

Calls a non-destructive version of `sv_setsv` if `dsb` is not the same as `ssv`. May evaluate arguments more than once.

```
void    SvSetSV_nosteal(SV* dsb, SV* ssv)
```

=for hackers Found in file `sv.h`

SvSHARE

Arranges for `sv` to be shared between threads if a suitable module has been loaded.

```
void    SvSHARE(SV* sv)
```

=for hackers Found in file `sv.h`

Memory Management

Copy The XSUB-writer’s interface to the C `memcpy` function. The `src` is the source, `dest` is the destination, `nitems` is the number of items, and `type` is the type. May fail on overlapping copies. See also `Move`.

```
void    Copy(void* src, void* dest, int nitems, type)
```

=for hackers Found in file `handy.h`

Move The XSUB-writer’s interface to the C `memmove` function. The `src` is the source, `dest` is the destination, `nitems` is the number of items, and `type` is the type. Can do overlapping moves. See also `Copy`.

```
void    Move(void* src, void* dest, int nitems, type)
```

=for hackers Found in file `handy.h`

- New** The XSUB-writer's interface to the C malloc function.
- ```
void New(int id, void* ptr, int nitems, type)
```
- =for hackers Found in file handy.h
- Newc** The XSUB-writer's interface to the C malloc function, with cast.
- ```
void Newc(int id, void* ptr, int nitems, type, cast)
```
- =for hackers Found in file handy.h
- NEWSV** Creates a new SV. A non-zero len parameter indicates the number of bytes of preallocated string space the SV should have. An extra byte for a trailing NUL is also reserved. (SvPOK is not set for the SV even if string space is allocated.) The reference count for the new SV is set to 1. id is an integer id between 0 and 1299 (used to identify leaks).
- ```
SV* NEWSV(int id, STRLEN len)
```
- =for hackers Found in file handy.h
- Newz** The XSUB-writer's interface to the C malloc function. The allocated memory is zeroed with memzero.
- ```
void Newz(int id, void* ptr, int nitems, type)
```
- =for hackers Found in file handy.h
- Poison** Fill up memory with a pattern (byte 0xAB over and over again) that hopefully catches attempts to access uninitialized memory.
- ```
void Poison(void* dest, int nitems, type)
```
- =for hackers Found in file handy.h
- Renew** The XSUB-writer's interface to the C realloc function.
- ```
void Renew(void* ptr, int nitems, type)
```
- =for hackers Found in file handy.h
- Renewc** The XSUB-writer's interface to the C realloc function, with cast.
- ```
void Renewc(void* ptr, int nitems, type, cast)
```
- =for hackers Found in file handy.h
- Safefree** The XSUB-writer's interface to the C free function.
- ```
void Safefree(void* ptr)
```
- =for hackers Found in file handy.h
- savepv** Perl's version of strdup(). Returns a pointer to a newly allocated string which is a duplicate of pv. The size of the string is determined by strlen(). The memory allocated for the new string can be freed with the Safefree() function.
- ```
char* savepv(const char* pv)
```
- =for hackers Found in file util.c
- savepvn** Perl's version of what strndup() would be if it existed. Returns a pointer to a newly allocated string which is a duplicate of the first len bytes from pv. The memory allocated for the new string can be freed with the Safefree() function.
- ```
char* savepvn(const char* pv, I32 len)
```
- =for hackers Found in file util.c

savesharedpv

A version of `savepv()` which allocates the duplicate string in memory which is shared between threads.

```
char*    savesharedpv(const char* pv)
```

=for hackers Found in file `util.c`

StructCopy

This is an architecture-independent macro to copy one structure to another.

```
void     StructCopy(type src, type dest, type)
```

=for hackers Found in file `handy.h`

Zero

The XSUB-writer's interface to the C `memzero` function. The `dest` is the destination, `nitems` is the number of items, and `type` is the type.

```
void     Zero(void* dest, int nitems, type)
```

=for hackers Found in file `handy.h`

Miscellaneous Functions**fbm_compile**

Analyses the string in order to make fast searches on it using `fbm_instr()` — the Boyer-Moore algorithm.

```
void     fbm_compile(SV* sv, U32 flags)
```

=for hackers Found in file `util.c`

fbm_instr Returns the location of the SV in the string delimited by `str` and `strend`. It returns `Nullch` if the string can't be found. The `sv` does not have to be `fbm_compiled`, but the search will not be as fast then.

```
char*    fbm_instr(unsigned char* big, unsigned char* bigend, SV* litt)
```

=for hackers Found in file `util.c`

form Takes a `sprintf`-style format pattern and conventional (non-SV) arguments and returns the formatted string.

```
(char *) Perl_form(pTHX_ const char* pat, ...)
```

can be used any place a string (`char *`) is required:

```
char * s = Perl_form("%d.%d", major, minor);
```

Uses a single private buffer so if you want to format several strings you must explicitly copy the earlier strings away (and free the copies when you are done).

```
char*    form(const char* pat, ...)
```

=for hackers Found in file `util.c`

getcwd_sv

Fill the `sv` with current working directory

```
int      getcwd_sv(SV* sv)
```

=for hackers Found in file `util.c`

strEQ

Test two strings to see if they are equal. Returns true or false.

```
bool     strEQ(char* s1, char* s2)
```

	=for hackers Found in file handy.h
strGE	Test two strings to see if the first, <i>s1</i> , is greater than or equal to the second, <i>s2</i> . Returns true or false. <pre>bool strGE(char* s1, char* s2)</pre> =for hackers Found in file handy.h
strGT	Test two strings to see if the first, <i>s1</i> , is greater than the second, <i>s2</i> . Returns true or false. <pre>bool strGT(char* s1, char* s2)</pre> =for hackers Found in file handy.h
strLE	Test two strings to see if the first, <i>s1</i> , is less than or equal to the second, <i>s2</i> . Returns true or false. <pre>bool strLE(char* s1, char* s2)</pre> =for hackers Found in file handy.h
strLT	Test two strings to see if the first, <i>s1</i> , is less than the second, <i>s2</i> . Returns true or false. <pre>bool strLT(char* s1, char* s2)</pre> =for hackers Found in file handy.h
strNE	Test two strings to see if they are different. Returns true or false. <pre>bool strNE(char* s1, char* s2)</pre> =for hackers Found in file handy.h
strnEQ	Test two strings to see if they are equal. The <i>len</i> parameter indicates the number of bytes to compare. Returns true or false. (A wrapper for <code>strncmp</code>). <pre>bool strnEQ(char* s1, char* s2, STRLEN len)</pre> =for hackers Found in file handy.h
strnNE	Test two strings to see if they are different. The <i>len</i> parameter indicates the number of bytes to compare. Returns true or false. (A wrapper for <code>strncmp</code>). <pre>bool strnNE(char* s1, char* s2, STRLEN len)</pre> =for hackers Found in file handy.h

Numeric functions

grok_bin converts a string representing a binary number to numeric form.

On entry *start* and **len* give the string to scan, **flags* gives conversion flags, and *result* should be NULL or a pointer to an NV. The scan stops at the end of the string, or the first invalid character. On return **len* is set to the length scanned string, and **flags* gives output flags.

If the value is \leq UV_MAX it is returned as a UV, the output flags are clear, and nothing is written to **result*. If the value is UV_MAX grok_bin returns UV_MAX, sets PERL_SCAN_GREATER_THAN_UV_MAX in the output flags, and writes the value to **result* (or the value is discarded if *result* is NULL).

The hex number may optionally be prefixed with "0b" or "b" unless PERL_SCAN_DISALLOW_PREFIX is set in **flags* on entry. If PERL_SCAN_ALLOW_UNDERSCORES is set in **flags* then the binary number may use '_' characters to separate digits.

```
UV      grok_bin(char* start, STRLEN* len, I32* flags, NV *result)
```

=for hackers Found in file numeric.c

grok_hex converts a string representing a hex number to numeric form.

On entry *start* and **len* give the string to scan, **flags* gives conversion flags, and *result* should be NULL or a pointer to an NV. The scan stops at the end of the string, or the first non-hex-digit character. On return **len* is set to the length scanned string, and **flags* gives output flags.

If the value is <= UV_MAX it is returned as a UV, the output flags are clear, and nothing is written to **result*. If the value is UV_MAX grok_hex returns UV_MAX, sets PERL_SCAN_GREATER_THAN_UV_MAX in the output flags, and writes the value to **result* (or the value is discarded if *result* is NULL).

The hex number may optionally be prefixed with "0x" or "x" unless PERL_SCAN_DISALLOW_PREFIX is set in **flags* on entry. If PERL_SCAN_ALLOW_UNDERSCORES is set in **flags* then the hex number may use '_' characters to separate digits.

```
UV      grok_hex(char* start, STRLEN* len, I32* flags, NV *result)
```

=for hackers Found in file numeric.c

grok_number

Recognise (or not) a number. The type of the number is returned (0 if unrecognised), otherwise it is a bit-ORed combination of IS_NUMBER_IN_UV, IS_NUMBER_GREATER_THAN_UV_MAX, IS_NUMBER_NOT_INT, IS_NUMBER_NEG, IS_NUMBER_INFINITY, IS_NUMBER_NAN (defined in perl.h).

If the value of the number can fit in UV, it is returned in the **valuep* IS_NUMBER_IN_UV will be set to indicate that **valuep* is valid, IS_NUMBER_IN_UV will never be set unless **valuep* is valid, but **valuep* may have been assigned to during processing even though IS_NUMBER_IN_UV is not set on return. If *valuep* is NULL, IS_NUMBER_IN_UV will be set for the same cases as when *valuep* is non-NULL, but no actual assignment (or SEGV) will occur.

IS_NUMBER_NOT_INT will be set with IS_NUMBER_IN_UV if trailing decimals were seen (in which case **valuep* gives the true value truncated to an integer), and IS_NUMBER_NEG if the number is negative (in which case **valuep* holds the absolute value). IS_NUMBER_IN_UV is not set if e notation was used or the number is larger than a UV.

```
int      grok_number(const char *pv, STRLEN len, UV *valuep)
```

=for hackers Found in file numeric.c

grok_numeric_radix

Scan and skip for a numeric decimal separator (radix).

```
bool      grok_numeric_radix(const char **sp, const char *send)
```

=for hackers Found in file numeric.c

grok_oct

```
UV      grok_oct(char* start, STRLEN* len, I32* flags, NV *result)
```

=for hackers Found in file numeric.c

scan_bin For backwards compatibility. Use grok_bin instead.

```
NV      scan_bin(char* start, STRLEN len, STRLEN* retlen)
```

=for hackers Found in file numeric.c

scan_hex

For backwards compatibility. Use `grok_hex` instead.

```
NV      scan_hex(char* start, STRLEN len, STRLEN* retlen)
```

=for hackers Found in file `numeric.c`

scan_oct For backwards compatibility. Use `grok_oct` instead.

```
NV      scan_oct(char* start, STRLEN len, STRLEN* retlen)
```

=for hackers Found in file `numeric.c`

Optree Manipulation Functions**cv_const_sv**

If `cv` is a constant sub eligible for inlining. returns the constant value returned by the sub. Otherwise, returns `NULL`.

Constant subs can be created with `newCONSTSUB` or as described in [Constant Functions in *perlsub*](#).

```
SV*      cv_const_sv(CV* cv)
```

=for hackers Found in file `op.c`

newCONSTSUB

Creates a constant sub equivalent to Perl sub `FOO () { 123 }` which is eligible for inlining at compile-time.

```
CV*      newCONSTSUB(HV* stash, char* name, SV* sv)
```

=for hackers Found in file `op.c`

newXS Used by `xsubpp` to hook up XSUBs as Perl subs.

=for hackers Found in file `op.c`

Stack Manipulation Macros

dMARK Declare a stack marker variable, `mark`, for the XSUB. See `MARK` and `dORIGMARK`.

```
dMARK;
```

=for hackers Found in file `pp.h`

dORIGMARK

Saves the original stack mark for the XSUB. See `ORIGMARK`.

```
dORIGMARK;
```

=for hackers Found in file `pp.h`

dSP Declares a local copy of perl's stack pointer for the XSUB, available via the `SP` macro. See `SP`.

```
dSP;
```

=for hackers Found in file `pp.h`

EXTEND Used to extend the argument stack for an XSUB's return values. Once used, guarantees that there is room for at least `nitems` to be pushed onto the stack.

```
void      EXTEND(SP, int nitems)
```

=for hackers Found in file `pp.h`

- MARK** Stack marker variable for the XSUB. See `dMARK`.
 =for hackers Found in file `pp.h`
- ORIGMARK**
 The original stack mark for the XSUB. See `dORIGMARK`.
 =for hackers Found in file `pp.h`
- POPi** Pops an integer off the stack.
 IV POPi
 =for hackers Found in file `pp.h`
- POPl** Pops a long off the stack.
 long POPl
 =for hackers Found in file `pp.h`
- POPn** Pops a double off the stack.
 NV POPn
 =for hackers Found in file `pp.h`
- POPp** Pops a string off the stack. Deprecated. New code should provide a `STRLEN n_a` and use `POPpx`.
 char* POPp
 =for hackers Found in file `pp.h`
- POPpbytex**
 Pops a string off the stack which must consist of bytes i.e. characters < 256. Requires a variable `STRLEN n_a` in scope.
 char* POPpbytex
 =for hackers Found in file `pp.h`
- POPpx** Pops a string off the stack. Requires a variable `STRLEN n_a` in scope.
 char* POPpx
 =for hackers Found in file `pp.h`
- POPs** Pops an SV off the stack.
 SV* POPs
 =for hackers Found in file `pp.h`
- PUSHi** Push an integer onto the stack. The stack must have room for this element. Handles ‘set’ magic. See `XPUSHi`.
 void PUSHi (IV iv)
 =for hackers Found in file `pp.h`
- PUSHMARK**
 Opening bracket for arguments on a callback. See `PUTBACK` and [perlcalloc](#).
 PUSHMARK;
 =for hackers Found in file `pp.h`

PUSHn Push a double onto the stack. The stack must have room for this element. Handles ‘set’ magic. See XPUSHn.

```
void    PUSHn(NV nv)
```

=for hackers Found in file pp.h

PUSHp Push a string onto the stack. The stack must have room for this element. The len indicates the length of the string. Handles ‘set’ magic. See XPUSHp.

```
void    PUSHp(char* str, STRLEN len)
```

=for hackers Found in file pp.h

PUSHs Push an SV onto the stack. The stack must have room for this element. Does not handle ‘set’ magic. See XPUSHs.

```
void    PUSHs(SV* sv)
```

=for hackers Found in file pp.h

PUSHu Push an unsigned integer onto the stack. The stack must have room for this element. See XPUSHu.

```
void    PUSHu(UV uv)
```

=for hackers Found in file pp.h

PUTBACK

Closing bracket for XSUB arguments. This is usually handled by xsubpp. See PUSHMARK and [perlcall](#) for other uses.

```
PUTBACK;
```

=for hackers Found in file pp.h

SP Stack pointer. This is usually handled by xsubpp. See dSP and SPAGAIN.

=for hackers Found in file pp.h

SPAGAIN

Refetch the stack pointer. Used after a callback. See [perlcall](#).

```
SPAGAIN;
```

=for hackers Found in file pp.h

XPUSHi Push an integer onto the stack, extending the stack if necessary. Handles ‘set’ magic. See PUSHi.

```
void    XPUSHi(IV iv)
```

=for hackers Found in file pp.h

XPUSHn Push a double onto the stack, extending the stack if necessary. Handles ‘set’ magic. See PUSHn.

```
void    XPUSHn(NV nv)
```

=for hackers Found in file pp.h

XPUSHp Push a string onto the stack, extending the stack if necessary. The len indicates the length of the string. Handles ‘set’ magic. See PUSHp.

```
void    XPUSHp(char* str, STRLEN len)
```

=for hackers Found in file pp.h

XPUSHs Push an SV onto the stack, extending the stack if necessary. Does not handle ‘set’ magic. See PUSHs.

```
void    XPUSHs(SV* sv)
```

=for hackers Found in file pp.h

XPUSHu Push an unsigned integer onto the stack, extending the stack if necessary. See PUSHu.

```
void    XPUSHu(UV uv)
```

=for hackers Found in file pp.h

XSRETURN

Return from XSUB, indicating number of items on the stack. This is usually handled by xsubpp.

```
void    XSRETURN(int nitems)
```

=for hackers Found in file XSUB.h

XSRETURN_IV

Return an integer from an XSUB immediately. Uses XST_mIV.

```
void    XSRETURN_IV(IV iv)
```

=for hackers Found in file XSUB.h

XSRETURN_NO

Return &PL_sv_no from an XSUB immediately. Uses XST_mNO.

```
XSRETURN_NO;
```

=for hackers Found in file XSUB.h

XSRETURN_NV

Return a double from an XSUB immediately. Uses XST_mNV.

```
void    XSRETURN_NV(NV nv)
```

=for hackers Found in file XSUB.h

XSRETURN_PV

Return a copy of a string from an XSUB immediately. Uses XST_mPV.

```
void    XSRETURN_PV(char* str)
```

=for hackers Found in file XSUB.h

XSRETURN_UNDEF

Return &PL_sv_undef from an XSUB immediately. Uses XST_mUNDEF.

```
XSRETURN_UNDEF;
```

=for hackers Found in file XSUB.h

XSRETURN_YES

Return &PL_sv_yes from an XSUB immediately. Uses XST_mYES.

```
XSRETURN_YES;
```

=for hackers Found in file XSUB.h

XST_mIV Place an integer into the specified position *pos* on the stack. The value is stored in a new mortal SV.

```
void    XST_mIV(int pos, IV iv)
```

=for hackers Found in file XSUB.h

XST_mNO

Place &PL_sv_no into the specified position `pos` on the stack.

```
void    XST_mNO(int pos)
```

=for hackers Found in file XSUB.h

XST_mNV

Place a double into the specified position `pos` on the stack. The value is stored in a new mortal SV.

```
void    XST_mNV(int pos, NV nv)
```

=for hackers Found in file XSUB.h

XST_mPV

Place a copy of a string into the specified position `pos` on the stack. The value is stored in a new mortal SV.

```
void    XST_mPV(int pos, char* str)
```

=for hackers Found in file XSUB.h

XST_mUNDEF

Place &PL_sv_undef into the specified position `pos` on the stack.

```
void    XST_mUNDEF(int pos)
```

=for hackers Found in file XSUB.h

XST_mYES

Place &PL_sv_yes into the specified position `pos` on the stack.

```
void    XST_mYES(int pos)
```

=for hackers Found in file XSUB.h

SV Flags

svtype An enum of flags for Perl types. These are found in the file **sv.h** in the **svtype** enum. Test these flags with the **SvTYPE** macro.

=for hackers Found in file sv.h

SVt_IV Integer type flag for scalars. See **svtype**.

=for hackers Found in file sv.h

SVt_NV Double type flag for scalars. See **svtype**.

=for hackers Found in file sv.h

SVt_PV Pointer type flag for scalars. See **svtype**.

=for hackers Found in file sv.h

SVt_PVAV

Type flag for arrays. See **svtype**.

=for hackers Found in file sv.h

SVt_PVCV

Type flag for code refs. See `svtype`.

=for hackers Found in file `sv.h`

SVt_PVHV

Type flag for hashes. See `svtype`.

=for hackers Found in file `sv.h`

SVt_PVMG

Type flag for blessed scalars. See `svtype`.

=for hackers Found in file `sv.h`

SV Manipulation Functions

get_sv Returns the SV of the specified Perl scalar. If `create` is set and the Perl variable does not exist then it will be created. If `create` is not set and the variable does not exist then NULL is returned.

NOTE: the `perl_` form of this function is deprecated.

```
SV*      get_sv(const char* name, I32 create)
```

=for hackers Found in file `perl.c`

looks_like_number

Test if the content of an SV looks like a number (or is a number). Inf and Infinity are treated as numbers (so will not issue a non-numeric warning), even if your `atof()` doesn't grok them.

```
I32      looks_like_number(SV* sv)
```

=for hackers Found in file `sv.c`

newRV_inc

Creates an RV wrapper for an SV. The reference count for the original SV is incremented.

```
SV*      newRV_inc(SV* sv)
```

=for hackers Found in file `sv.h`

newRV_noinc

Creates an RV wrapper for an SV. The reference count for the original SV is **not** incremented.

```
SV*      newRV_noinc(SV *sv)
```

=for hackers Found in file `sv.c`

newSV

Create a new null SV, or if `len > 0`, create a new empty SVt_PV type SV with an initial PV allocation of `len+1`. Normally accessed via the `NEWSV` macro.

```
SV*      newSV(STRLEN len)
```

=for hackers Found in file `sv.c`

newSViv

Creates a new SV and copies an integer into it. The reference count for the SV is set to 1.

```
SV*      newSViv(IV i)
```

=for hackers Found in file `sv.c`

newSVnv

Creates a new SV and copies a floating point value into it. The reference count for the SV is set to 1.

SV* newSVnv(NV n)

=for hackers Found in file sv.c

newSVpv Creates a new SV and copies a string into it. The reference count for the SV is set to 1. If len is zero, Perl will compute the length using strlen(). For efficiency, consider using newSVpvn instead.

SV* newSVpv(const char* s, STRLEN len)

=for hackers Found in file sv.c

newSVpvf

Creates a new SV and initializes it with the string formatted like sprintf.

SV* newSVpvf(const char* pat, ...)

=for hackers Found in file sv.c

newSVpvn

Creates a new SV and copies a string into it. The reference count for the SV is set to 1. Note that if len is zero, Perl will create a zero length string. You are responsible for ensuring that the source string is at least len bytes long.

SV* newSVpvn(const char* s, STRLEN len)

=for hackers Found in file sv.c

newSVpvn_share

Creates a new SV with its SvPVX pointing to a shared string in the string table. If the string does not already exist in the table, it is created first. Turns on READONLY and FAKE. The string's hash is stored in the UV slot of the SV; if the hash parameter is non-zero, that value is used; otherwise the hash is computed. The idea here is that as the string table is used for shared hash keys these strings will have SvPVX == HeKEY and hash lookup will avoid string compare.

SV* newSVpvn_share(const char* s, I32 len, U32 hash)

=for hackers Found in file sv.c

newSVrv Creates a new SV for the RV, rv, to point to. If rv is not an RV then it will be upgraded to one. If classname is non-null then the new SV will be blessed in the specified package. The new SV is returned and its reference count is 1.

SV* newSVrv(SV* rv, const char* classname)

=for hackers Found in file sv.c

newSVsv Creates a new SV which is an exact duplicate of the original SV. (Uses sv_setsv).

SV* newSVsv(SV* old)

=for hackers Found in file sv.c

newSVuv Creates a new SV and copies an unsigned integer into it. The reference count for the SV is set to 1.

SV* newSVuv(UV u)

=for hackers Found in file sv.c

new_vstring

Returns a pointer to the next character after the parsed vstring, as well as updating the passed in sv.

Function must be called like

```
sv = NEWSV(92,5);
s = new_vstring(s,sv);
```

The sv must already be large enough to store the vstring passed in.

```
char*    new_vstring(char *vstr, SV *sv)
```

=for hackers Found in file util.c

SvCUR Returns the length of the string which is in the SV. See SvLEN.

```
STRLEN   SvCUR(SV* sv)
```

=for hackers Found in file sv.h

SvCUR_set

Set the length of the string which is in the SV. See SvCUR.

```
void      SvCUR_set(SV* sv, STRLEN len)
```

=for hackers Found in file sv.h

SvEND Returns a pointer to the last character in the string which is in the SV. See SvCUR. Access the character as `*(SvEND(sv))`.

```
char*     SvEND(SV* sv)
```

=for hackers Found in file sv.h

SvGROW

Expands the character buffer in the SV so that it has room for the indicated number of bytes (remember to reserve space for an extra trailing NUL character). Calls `sv_grow` to perform the expansion if necessary. Returns a pointer to the character buffer.

```
char *    SvGROW(SV* sv, STRLEN len)
```

=for hackers Found in file sv.h

SvIOK Returns a boolean indicating whether the SV contains an integer.

```
bool      SvIOK(SV* sv)
```

=for hackers Found in file sv.h

SvIOKp Returns a boolean indicating whether the SV contains an integer. Checks the **private** setting. Use SvIOK.

```
bool      SvIOKp(SV* sv)
```

=for hackers Found in file sv.h

SvIOK_notUV

Returns a boolean indicating whether the SV contains a signed integer.

```
void      SvIOK_notUV(SV* sv)
```

=for hackers Found in file sv.h

SvIOK_off

Unsets the IV status of an SV.

```
void      SvIOK_off(SV* sv)
```

=for hackers Found in file sv.h

SvIOK_on

Tells an SV that it is an integer.

```
void      SvIOK_on(SV* sv)
```

=for hackers Found in file sv.h

SvIOK_only

Tells an SV that it is an integer and disables all other OK bits.

```
void      SvIOK_only(SV* sv)
```

=for hackers Found in file sv.h

SvIOK_only_UV

Tells and SV that it is an unsigned integer and disables all other OK bits.

```
void      SvIOK_only_UV(SV* sv)
```

=for hackers Found in file sv.h

SvIOK_UV

Returns a boolean indicating whether the SV contains an unsigned integer.

```
void      SvIOK_UV(SV* sv)
```

=for hackers Found in file sv.h

SvIV

Coerces the given SV to an integer and returns it. See `SvIVx` for a version which guarantees to evaluate sv only once.

```
IV        SvIV(SV* sv)
```

=for hackers Found in file sv.h

SvIVx

Coerces the given SV to an integer and returns it. Guarantees to evaluate sv only once. Use the more efficient `SvIV` otherwise.

```
IV        SvIVx(SV* sv)
```

=for hackers Found in file sv.h

SvIVX

Returns the raw value in the SV's IV slot, without checks or conversions. Only use when you are sure `SvIOK` is true. See also `SvIV()`.

```
IV        SvIVX(SV* sv)
```

=for hackers Found in file sv.h

SvLEN

Returns the size of the string buffer in the SV, not including any part attributable to `SvOOK`. See `SvCUR`.

```
STRLEN    SvLEN(SV* sv)
```

=for hackers Found in file sv.h

SvNIOK

Returns a boolean indicating whether the SV contains a number, integer or double.

```
bool      SvNIOK(SV* sv)
```

=for hackers Found in file sv.h

SvNIOKp

Returns a boolean indicating whether the SV contains a number, integer or double. Checks the **private** setting. Use `SvNIOK`.

```
bool      SvNIOKp(SV* sv)
```

=for hackers Found in file sv.h

SvNIOK_off

Unsets the NV/IV status of an SV.

```
void      SvNIOK_off(SV* sv)
```

=for hackers Found in file sv.h

SvNOK Returns a boolean indicating whether the SV contains a double.

```
bool      SvNOK(SV* sv)
```

=for hackers Found in file sv.h

SvNOKp Returns a boolean indicating whether the SV contains a double. Checks the **private** setting. Use SvNOK.

```
bool      SvNOKp(SV* sv)
```

=for hackers Found in file sv.h

SvNOK_off

Unsets the NV status of an SV.

```
void      SvNOK_off(SV* sv)
```

=for hackers Found in file sv.h

SvNOK_on

Tells an SV that it is a double.

```
void      SvNOK_on(SV* sv)
```

=for hackers Found in file sv.h

SvNOK_only

Tells an SV that it is a double and disables all other OK bits.

```
void      SvNOK_only(SV* sv)
```

=for hackers Found in file sv.h

SvNV Coerce the given SV to a double and return it. See SvNVx for a version which guarantees to evaluate sv only once.

```
NV        SvNV(SV* sv)
```

=for hackers Found in file sv.h

SvNVX Returns the raw value in the SV's NV slot, without checks or conversions. Only use when you are sure SvNOK is true. See also SvNV().

```
NV        SvNVX(SV* sv)
```

=for hackers Found in file sv.h

SvNVx Coerces the given SV to a double and returns it. Guarantees to evaluate sv only once. Use the more efficient SvNV otherwise.

```
NV        SvNVx(SV* sv)
```

=for hackers Found in file sv.h

SvOK Returns a boolean indicating whether the value is an SV.

```
bool      SvOK(SV* sv)
```

=for hackers Found in file sv.h

SvOOK Returns a boolean indicating whether the SvIVX is a valid offset value for the SvPVX. This hack is used internally to speed up removal of characters from the beginning of a SvPV. When SvOOK is true, then the start of the allocated string buffer is really (SvPVX – SvIVX).

```
bool    SvOOK(SV* sv)
```

=for hackers Found in file sv.h

SvPOK Returns a boolean indicating whether the SV contains a character string.

```
bool    SvPOK(SV* sv)
```

=for hackers Found in file sv.h

SvPOKp Returns a boolean indicating whether the SV contains a character string. Checks the **private** setting. Use SvPOK.

```
bool    SvPOKp(SV* sv)
```

=for hackers Found in file sv.h

SvPOK_off

Unsets the PV status of an SV.

```
void    SvPOK_off(SV* sv)
```

=for hackers Found in file sv.h

SvPOK_on

Tells an SV that it is a string.

```
void    SvPOK_on(SV* sv)
```

=for hackers Found in file sv.h

SvPOK_only

Tells an SV that it is a string and disables all other OK bits. Will also turn off the UTF8 status.

```
void    SvPOK_only(SV* sv)
```

=for hackers Found in file sv.h

SvPOK_only_UTF8

Tells an SV that it is a string and disables all other OK bits, and leaves the UTF8 status as it was.

```
void    SvPOK_only_UTF8(SV* sv)
```

=for hackers Found in file sv.h

SvPV

Returns a pointer to the string in the SV, or a stringified form of the SV if the SV does not contain a string. The SV may cache the stringified version becoming SvPOK. Handles 'get' magic. See also SvPVx for a version which guarantees to evaluate sv only once.

```
char*    SvPV(SV* sv, STRLEN len)
```

=for hackers Found in file sv.h

SvPVbyte

Like SvPV, but converts sv to byte representation first if necessary.

```
char*    SvPVbyte(SV* sv, STRLEN len)
```

=for hackers Found in file sv.h

SvPVbyte

Like `SvPV`, but converts `sv` to byte representation first if necessary. Guarantees to evaluate `sv` only once; use the more efficient `SvPVbyte` otherwise.

```
char* SvPVbyte(SV* sv, STRLEN len)
```

=for hackers Found in file `sv.h`

SvPVbyte_force

Like `SvPV_force`, but converts `sv` to byte representation first if necessary. Guarantees to evaluate `sv` only once; use the more efficient `SvPVbyte_force` otherwise.

```
char* SvPVbyte_force(SV* sv, STRLEN len)
```

=for hackers Found in file `sv.h`

SvPVbyte_force

Like `SvPV_force`, but converts `sv` to byte representation first if necessary.

```
char* SvPVbyte_force(SV* sv, STRLEN len)
```

=for hackers Found in file `sv.h`

SvPVbyte_nolen

Like `SvPV_nolen`, but converts `sv` to byte representation first if necessary.

```
char* SvPVbyte_nolen(SV* sv)
```

=for hackers Found in file `sv.h`

SvPVutf8 Like `SvPV`, but converts `sv` to utf8 first if necessary.

```
char* SvPVutf8(SV* sv, STRLEN len)
```

=for hackers Found in file `sv.h`

SvPVutf8x

Like `SvPV`, but converts `sv` to utf8 first if necessary. Guarantees to evaluate `sv` only once; use the more efficient `SvPVutf8` otherwise.

```
char* SvPVutf8x(SV* sv, STRLEN len)
```

=for hackers Found in file `sv.h`

SvPVutf8x_force

Like `SvPV_force`, but converts `sv` to utf8 first if necessary. Guarantees to evaluate `sv` only once; use the more efficient `SvPVutf8_force` otherwise.

```
char* SvPVutf8x_force(SV* sv, STRLEN len)
```

=for hackers Found in file `sv.h`

SvPVutf8_force

Like `SvPV_force`, but converts `sv` to utf8 first if necessary.

```
char* SvPVutf8_force(SV* sv, STRLEN len)
```

=for hackers Found in file `sv.h`

SvPVutf8_nolen

Like `SvPV_nolen`, but converts `sv` to utf8 first if necessary.

```
char* SvPVutf8_nolen(SV* sv)
```

=for hackers Found in file `sv.h`

SvPVx A version of `SvPV` which guarantees to evaluate `sv` only once.

```
char* SvPVx(SV* sv, STRLEN len)
```

=for hackers Found in file `sv.h`

SvPVX Returns a pointer to the physical string in the SV. The SV must contain a string.

```
char* SvPVX(SV* sv)
```

=for hackers Found in file `sv.h`

SvPV_force

Like `SvPV` but will force the SV into containing just a string (`SvPOK_only`). You want force if you are going to update the `SvPVX` directly.

```
char* SvPV_force(SV* sv, STRLEN len)
```

=for hackers Found in file `sv.h`

SvPV_force_nomg

Like `SvPV` but will force the SV into containing just a string (`SvPOK_only`). You want force if you are going to update the `SvPVX` directly. Doesn't process magic.

```
char* SvPV_force_nomg(SV* sv, STRLEN len)
```

=for hackers Found in file `sv.h`

SvPV_nolen

Returns a pointer to the string in the SV, or a stringified form of the SV if the SV does not contain a string. The SV may cache the stringified form becoming `SvPOK`. Handles 'get' magic.

```
char* SvPV_nolen(SV* sv)
```

=for hackers Found in file `sv.h`

SvREFCNT

Returns the value of the object's reference count.

```
U32 SvREFCNT(SV* sv)
```

=for hackers Found in file `sv.h`

SvREFCNT_dec

Decrements the reference count of the given SV.

```
void SvREFCNT_dec(SV* sv)
```

=for hackers Found in file `sv.h`

SvREFCNT_inc

Increments the reference count of the given SV.

```
SV* SvREFCNT_inc(SV* sv)
```

=for hackers Found in file `sv.h`

SvROK Tests if the SV is an RV.

```
bool SvROK(SV* sv)
```

=for hackers Found in file `sv.h`

SvROK_off

Unsets the RV status of an SV.

```
void      SvROK_off(SV* sv)
```

=for hackers Found in file sv.h

SvROK_on

Tells an SV that it is an RV.

```
void      SvROK_on(SV* sv)
```

=for hackers Found in file sv.h

SvRV

Dereferences an RV to return the SV.

```
SV*       SvRV(SV* sv)
```

=for hackers Found in file sv.h

SvSTASH

Returns the stash of the SV.

```
HV*       SvSTASH(SV* sv)
```

=for hackers Found in file sv.h

SvTAINT Taints an SV if tainting is enabled

```
void      SvTAINT(SV* sv)
```

=for hackers Found in file sv.h

SvTAINTED

Checks to see if an SV is tainted. Returns TRUE if it is, FALSE if not.

```
bool      SvTAINTED(SV* sv)
```

=for hackers Found in file sv.h

SvTAINTED_off

Untaints an SV. Be *very* careful with this routine, as it short-circuits some of Perl's fundamental security features. XS module authors should not use this function unless they fully understand all the implications of unconditionally untainting the value. Untainting should be done in the standard perl fashion, via a carefully crafted regexp, rather than directly untainting variables.

```
void      SvTAINTED_off(SV* sv)
```

=for hackers Found in file sv.h

SvTAINTED_on

Marks an SV as tainted.

```
void      SvTAINTED_on(SV* sv)
```

=for hackers Found in file sv.h

SvTRUE

Returns a boolean indicating whether Perl would evaluate the SV as true or false, defined or undefined. Does not handle 'get' magic.

```
bool      SvTRUE(SV* sv)
```

=for hackers Found in file sv.h

SvTYPE Returns the type of the SV. See `svtype`.

```
svtype SvTYPE(SV* sv)
```

=for hackers Found in file `sv.h`

SvUNLOCK

Releases a mutual exclusion lock on `sv` if a suitable module has been loaded.

```
void SvUNLOCK(SV* sv)
```

=for hackers Found in file `sv.h`

SvUOK Returns a boolean indicating whether the SV contains an unsigned integer.

```
void SvUOK(SV* sv)
```

=for hackers Found in file `sv.h`

SvUPGRADE

Used to upgrade an SV to a more complex form. Uses `sv_upgrade` to perform the upgrade if necessary. See `svtype`.

```
void SvUPGRADE(SV* sv, svtype type)
```

=for hackers Found in file `sv.h`

SvUTF8 Returns a boolean indicating whether the SV contains UTF-8 encoded data.

```
void SvUTF8(SV* sv)
```

=for hackers Found in file `sv.h`

SvUTF8_off

Unsets the UTF8 status of an SV.

```
void SvUTF8_off(SV *sv)
```

=for hackers Found in file `sv.h`

SvUTF8_on

Turn on the UTF8 status of an SV (the data is not changed, just the flag). Do not use frivolously.

```
void SvUTF8_on(SV *sv)
```

=for hackers Found in file `sv.h`

SvUV Coerces the given SV to an unsigned integer and returns it. See `SvUVx` for a version which guarantees to evaluate `sv` only once.

```
UV SvUV(SV* sv)
```

=for hackers Found in file `sv.h`

SvUVX Returns the raw value in the SV's UV slot, without checks or conversions. Only use when you are sure `SvIOK` is true. See also `SvUV()`.

```
UV SvUVX(SV* sv)
```

=for hackers Found in file `sv.h`

SvUVx Coerces the given SV to an unsigned integer and returns it. Guarantees to evaluate `sv` only once. Use the more efficient `SvUV` otherwise.

```
UV SvUVx(SV* sv)
```

=for hackers Found in file `sv.h`

sv_2bool

This function is only called on magical items, and is only used by `sv_true()` or its macro equivalent.

```
bool      sv_2bool(SV* sv)
```

=for hackers Found in file `sv.c`

sv_2cv

Using various gambits, try to get a CV from an SV; in addition, try if possible to set `*st` and `*gvp` to the stash and GV associated with it.

```
CV*       sv_2cv(SV* sv, HV** st, GV** gvp, I32 lref)
```

=for hackers Found in file `sv.c`

sv_2io

Using various gambits, try to get an IO from an SV: the IO slot if its a GV; or the recursive result if we're an RV; or the IO slot of the symbol named after the PV if we're a string.

```
IO*       sv_2io(SV* sv)
```

=for hackers Found in file `sv.c`

sv_2iv

Return the integer value of an SV, doing any necessary string conversion, magic etc. Normally used via the `SvIV(sv)` and `SvIVx(sv)` macros.

```
IV        sv_2iv(SV* sv)
```

=for hackers Found in file `sv.c`

sv_2mortal

Marks an existing SV as mortal. The SV will be destroyed "soon", either by an explicit call to `FREETMPS`, or by an implicit call at places such as statement boundaries. See also `sv_newmortal` and `sv_mortalcopy`.

```
SV*       sv_2mortal(SV* sv)
```

=for hackers Found in file `sv.c`

sv_2nv

Return the num value of an SV, doing any necessary string or integer conversion, magic etc. Normally used via the `SvNV(sv)` and `SvNVx(sv)` macros.

```
NV        sv_2nv(SV* sv)
```

=for hackers Found in file `sv.c`

sv_2pvbyte

Return a pointer to the byte-encoded representation of the SV, and set `*lp` to its length. May cause the SV to be downgraded from UTF8 as a side-effect.

Usually accessed via the `SvPVbyte` macro.

```
char*     sv_2pvbyte(SV* sv, STRLEN* lp)
```

=for hackers Found in file `sv.c`

sv_2pvbyte_nolen

Return a pointer to the byte-encoded representation of the SV. May cause the SV to be downgraded from UTF8 as a side-effect.

Usually accessed via the `SvPVbyte_nolen` macro.

```
char*     sv_2pvbyte_nolen(SV* sv)
```

=for hackers Found in file `sv.c`

sv_2pvutf8

Return a pointer to the UTF8-encoded representation of the SV, and set *lp to its length. May cause the SV to be upgraded to UTF8 as a side-effect.

Usually accessed via the SvPVutf8 macro.

```
char*    sv_2pvutf8(SV* sv, STRLEN* lp)
```

=for hackers Found in file sv.c

sv_2pvutf8_nolen

Return a pointer to the UTF8-encoded representation of the SV. May cause the SV to be upgraded to UTF8 as a side-effect.

Usually accessed via the SvPVutf8_nolen macro.

```
char*    sv_2pvutf8_nolen(SV* sv)
```

=for hackers Found in file sv.c

sv_2pv_flags

Returns a pointer to the string value of an SV, and sets *lp to its length. If flags includes SV_GMAGIC, does an mg_get() first. Coerces sv to a string if necessary. Normally invoked via the SvPV_flags macro. sv_2pv() and sv_2pv_nomg usually end up here too.

```
char*    sv_2pv_flags(SV* sv, STRLEN* lp, I32 flags)
```

=for hackers Found in file sv.c

sv_2pv_nolen

Like sv_2pv(), but doesn't return the length too. You should usually use the macro wrapper SvPV_nolen(sv) instead.

```
char*    sv_2pv_nolen(SV* sv)
```

=for hackers Found in file sv.c

sv_2uv Return the unsigned integer value of an SV, doing any necessary string conversion, magic etc. Normally used via the SvUV(sv) and SvUVx(sv) macros.

```
UV        sv_2uv(SV* sv)
```

=for hackers Found in file sv.c

sv_backoff

Remove any string offset. You should normally use the SvOOK_off macro wrapper instead.

```
int        sv_backoff(SV* sv)
```

=for hackers Found in file sv.c

sv_bless Blesses an SV into a specified package. The SV must be an RV. The package must be designated by its stash (see gv_stashpv()). The reference count of the SV is unaffected.

```
SV*        sv_bless(SV* sv, HV* stash)
```

=for hackers Found in file sv.c

sv_catpv Concatenates the string onto the end of the string which is in the SV. If the SV has the UTF8 status set, then the bytes appended should be valid UTF8. Handles 'get' magic, but not 'set' magic. See sv_catpv_mg.

```
void        sv_catpv(SV* sv, const char* ptr)
```

=for hackers Found in file sv.c

sv_catpvf

Processes its arguments like `sprintf` and appends the formatted output to an SV. If the appended data contains "wide" characters (including, but not limited to, SVs with a UTF-8 PV formatted with `%s`, and characters 255 formatted with `%c`), the original SV might get upgraded to UTF-8. Handles 'get' magic, but not 'set' magic. `SvSETMAGIC()` must typically be called after calling this function to handle 'set' magic.

```
void    sv_catpvf(SV* sv, const char* pat, ...)
```

=for hackers Found in file sv.c

sv_catpvf_mg

Like `sv_catpvf`, but also handles 'set' magic.

```
void    sv_catpvf_mg(SV *sv, const char* pat, ...)
```

=for hackers Found in file sv.c

sv_catpv

Concatenates the string onto the end of the string which is in the SV. The `len` indicates number of bytes to copy. If the SV has the UTF8 status set, then the bytes appended should be valid UTF8. Handles 'get' magic, but not 'set' magic. See `sv_catpv_mg`.

```
void    sv_catpv(SV* sv, const char* ptr, STRLEN len)
```

=for hackers Found in file sv.c

sv_catpv_flags

Concatenates the string onto the end of the string which is in the SV. The `len` indicates number of bytes to copy. If the SV has the UTF8 status set, then the bytes appended should be valid UTF8. If `flags` has `SV_GMAGIC` bit set, will `mg_get` on `dsv` if appropriate, else not. `sv_catpv` and `sv_catpv_nomg` are implemented in terms of this function.

```
void    sv_catpv_flags(SV* sv, const char* ptr, STRLEN len, I32 flag)
```

=for hackers Found in file sv.c

sv_catpv_mg

Like `sv_catpv`, but also handles 'set' magic.

```
void    sv_catpv_mg(SV *sv, const char *ptr, STRLEN len)
```

=for hackers Found in file sv.c

sv_catpv_mg

Like `sv_catpv`, but also handles 'set' magic.

```
void    sv_catpv_mg(SV *sv, const char *ptr)
```

=for hackers Found in file sv.c

sv_catsv Concatenates the string from SV `ssv` onto the end of the string in SV `dsv`. Modifies `dsv` but not `ssv`. Handles 'get' magic, but not 'set' magic. See `sv_catsv_mg`.

```
void    sv_catsv(SV* dsv, SV* ssv)
```

=for hackers Found in file sv.c

sv_catsv_flags

Concatenates the string from SV `ssv` onto the end of the string in SV `dsv`. Modifies `dsv` but not `ssv`. If `flags` has `SV_GMAGIC` bit set, will `mg_get` on the SVs if appropriate, else not. `sv_catsv` and `sv_catsv_nomg` are implemented in terms of this function.

```
void    sv_catsv_flags(SV* dsv, SV* ssv, I32 flags)
```

=for hackers Found in file sv.c

sv_catsv_mg

Like `sv_catsv`, but also handles ‘set’ magic.

```
void    sv_catsv_mg(SV *dstr, SV *sstr)
```

=for hackers Found in file sv.c

sv_chop Efficient removal of characters from the beginning of the string buffer. `SvPOK(sv)` must be true and the `ptr` must be a pointer to somewhere inside the string buffer. The `ptr` becomes the first character of the adjusted string. Uses the “OOK hack”.

```
void    sv_chop(SV* sv, char* ptr)
```

=for hackers Found in file sv.c

sv_clear Clear an SV: call any destructors, free up any memory used by the body, and free the body itself. The SV’s head is *not* freed, although its type is set to all 1’s so that it won’t inadvertently be assumed to be live during global destruction etc. This function should only be called when `REFCNT` is zero. Most of the time you’ll want to call `sv_free()` (or its macro wrapper `SvREFCNT_dec`) instead.

```
void    sv_clear(SV* sv)
```

=for hackers Found in file sv.c

sv_cmp Compares the strings in two SVs. Returns -1, 0, or 1 indicating whether the string in `sv1` is less than, equal to, or greater than the string in `sv2`. Is UTF-8 and ‘use bytes’ aware, handles get magic, and will coerce its args to strings if necessary. See also `sv_cmp_locale`.

```
I32     sv_cmp(SV* sv1, SV* sv2)
```

=for hackers Found in file sv.c

sv_cmp_locale

Compares the strings in two SVs in a locale-aware manner. Is UTF-8 and ‘use bytes’ aware, handles get magic, and will coerce its args to strings if necessary. See also `sv_cmp_locale`. See also `sv_cmp`.

```
I32     sv_cmp_locale(SV* sv1, SV* sv2)
```

=for hackers Found in file sv.c

sv_collxfrm

Add Collate Transform magic to an SV if it doesn’t already have it.

Any scalar variable may carry `PERL_MAGIC_collxfrm` magic that contains the scalar data of the variable, but transformed to such a format that a normal memory comparison can be used to compare the data according to the locale settings.

```
char*    sv_collxfrm(SV* sv, STRLEN* npx)
```

=for hackers Found in file sv.c

sv_copypv

Copies a stringified representation of the source SV into the destination SV. Automatically performs any necessary `mg_get` and coercion of numeric values into strings. Guaranteed to preserve UTF-8 flag even from overloaded objects. Similar in nature to `sv_2pv[_flags]` but operates directly on an SV instead of just the string. Mostly uses `sv_2pv_flags` to do its work, except when that would lose the UTF-8’ness of the PV.

- `void sv_copypv(SV* dsv, SV* ssv)`
 =for hackers Found in file sv.c
- sv_dec** Auto-decrement of the value in the SV, doing string to numeric conversion if necessary. Handles 'get' magic.
`void sv_dec(SV* sv)`
 =for hackers Found in file sv.c
- sv_derived_from**
 Returns a boolean indicating whether the SV is derived from the specified class. This is the function that implements `UNIVERSAL::isa`. It works for class names as well as for objects.
`bool sv_derived_from(SV* sv, const char* name)`
 =for hackers Found in file universal.c
- sv_eq** Returns a boolean indicating whether the strings in the two SVs are identical. Is UTF-8 and 'use bytes' aware, handles get magic, and will coerce its args to strings if necessary.
`I32 sv_eq(SV* sv1, SV* sv2)`
 =for hackers Found in file sv.c
- sv_force_normal**
 Undo various types of fakery on an SV: if the PV is a shared string, make a private copy; if we're a ref, stop refing; if we're a glob, downgrade to an xpvmg. See also `sv_force_normal_flags`.
`void sv_force_normal(SV *sv)`
 =for hackers Found in file sv.c
- sv_force_normal_flags**
 Undo various types of fakery on an SV: if the PV is a shared string, make a private copy; if we're a ref, stop refing; if we're a glob, downgrade to an xpvmg. The `flags` parameter gets passed to `sv_unref_flags()` when unrefing. `sv_force_normal` calls this function with `flags` set to 0.
`void sv_force_normal_flags(SV *sv, U32 flags)`
 =for hackers Found in file sv.c
- sv_free** Decrement an SV's reference count, and if it drops to zero, call `sv_clear` to invoke destructors and free up any memory used by the body; finally, deallocate the SV's head itself. Normally called via a wrapper macro `SvREFCNT_dec`.
`void sv_free(SV* sv)`
 =for hackers Found in file sv.c
- sv_gets** Get a line from the filehandle and store it into the SV, optionally appending to the currently-stored string.
`char* sv_gets(SV* sv, PerlIO* fp, I32 append)`
 =for hackers Found in file sv.c
- sv_grow** Expands the character buffer in the SV. If necessary, uses `sv_unref` and upgrades the SV to `SVt_PV`. Returns a pointer to the character buffer. Use the `SvGROW` wrapper instead.
`char* sv_grow(SV* sv, STRLEN newlen)`
 =for hackers Found in file sv.c

sv_inc Auto-increment of the value in the SV, doing string to numeric conversion if necessary. Handles ‘get’ magic.

```
void sv_inc(SV* sv)
```

=for hackers Found in file sv.c

sv_insert Inserts a string at the specified offset/length within the SV. Similar to the Perl `substr()` function.

```
void sv_insert(SV* bigsv, STRLEN offset, STRLEN len, char* little,
```

=for hackers Found in file sv.c

sv_isa Returns a boolean indicating whether the SV is blessed into the specified class. This does not check for subtypes; use `sv_derived_from` to verify an inheritance relationship.

```
int sv_isa(SV* sv, const char* name)
```

=for hackers Found in file sv.c

sv_isobject

Returns a boolean indicating whether the SV is an RV pointing to a blessed object. If the SV is not an RV, or if the object is not blessed, then this will return false.

```
int sv_isobject(SV* sv)
```

=for hackers Found in file sv.c

sv_iv A private implementation of the `SvIVx` macro for compilers which can’t cope with complex macro expressions. Always use the macro instead.

```
IV sv_iv(SV* sv)
```

=for hackers Found in file sv.c

sv_len Returns the length of the string in the SV. Handles magic and type coercion. See also `SvCUR`, which gives raw access to the `xpv_cur` slot.

```
STRLEN sv_len(SV* sv)
```

=for hackers Found in file sv.c

sv_len_utf8

Returns the number of characters in the string in an SV, counting wide UTF8 bytes as a single character. Handles magic and type coercion.

```
STRLEN sv_len_utf8(SV* sv)
```

=for hackers Found in file sv.c

sv_magic Adds magic to an SV. First upgrades `sv` to type `SVt_PVMG` if necessary, then adds a new magic item of type `how` to the head of the magic list.

```
void sv_magic(SV* sv, SV* obj, int how, const char* name, I32 namlen)
```

=for hackers Found in file sv.c

sv_magicext

Adds magic to an SV, upgrading it if necessary. Applies the supplied vtable and returns pointer to the magic added.

Note that `sv_magicext` will allow things that `sv_magic` will not. In particular you can add magic to `SvREADONLY` SVs and more than one instance of the same ‘how’

If `namelen` is greater than zero then a `savepvn()` copy of `name` is stored, if `namelen` is zero

then name is stored as-is and – as another special case – if (name && namelen == HEF_SVKEY) then name is assumed to contain an SV* and has its REFCNT incremented

(This is now used as a subroutine by sv_magic.)

```
MAGIC * sv_magicext(SV* sv, SV* obj, int how, MGVTBL *vtbl, const cha
```

=for hackers Found in file sv.c

sv_mortalcopy

Creates a new SV which is a copy of the original SV (using sv_setsv). The new SV is marked as mortal. It will be destroyed "soon", either by an explicit call to FREETMPS, or by an implicit call at places such as statement boundaries. See also sv_newmortal and sv_2mortal.

```
SV*      sv_mortalcopy(SV* oldsv)
```

=for hackers Found in file sv.c

sv_newmortal

Creates a new null SV which is mortal. The reference count of the SV is set to 1. It will be destroyed "soon", either by an explicit call to FREETMPS, or by an implicit call at places such as statement boundaries. See also sv_mortalcopy and sv_2mortal.

```
SV*      sv_newmortal()
```

=for hackers Found in file sv.c

sv_newref

Increment an SV's reference count. Use the SvREFCNT_inc() wrapper instead.

```
SV*      sv_newref(SV* sv)
```

=for hackers Found in file sv.c

sv_nolocking

Dummy routine which "locks" an SV when there is no locking module present. Exists to avoid test for a NULL function pointer and because it could potentially warn under some level of strict-ness.

```
void      sv_nolocking(SV *)
```

=for hackers Found in file util.c

sv_nosharing

Dummy routine which "shares" an SV when there is no sharing module present. Exists to avoid test for a NULL function pointer and because it could potentially warn under some level of strict-ness.

```
void      sv_nosharing(SV *)
```

=for hackers Found in file util.c

sv_nounlocking

Dummy routine which "unlocks" an SV when there is no locking module present. Exists to avoid test for a NULL function pointer and because it could potentially warn under some level of strict-ness.

```
void      sv_nounlocking(SV *)
```

=for hackers Found in file util.c

sv_nv

A private implementation of the SvNVx macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

NV `sv_nv(SV* sv)`

=for hackers Found in file sv.c

`sv_pos_b2u`

Converts the value pointed to by `offsetp` from a count of bytes from the start of the string, to a count of the equivalent number of UTF8 chars. Handles magic and type coercion.

void `sv_pos_b2u(SV* sv, I32* offsetp)`

=for hackers Found in file sv.c

`sv_pos_u2b`

Converts the value pointed to by `offsetp` from a count of UTF8 chars from the start of the string, to a count of the equivalent number of bytes; if `lenp` is non-zero, it does the same to `lenp`, but this time starting from the offset, rather than from the start of the string. Handles magic and type coercion.

void `sv_pos_u2b(SV* sv, I32* offsetp, I32* lenp)`

=for hackers Found in file sv.c

`sv_pv` Use the `SvPV_nolen` macro instead

char* `sv_pv(SV *sv)`

=for hackers Found in file sv.c

`sv_pvbyte`

Use `SvPVbyte_nolen` instead.

char* `sv_pvbyte(SV *sv)`

=for hackers Found in file sv.c

`sv_pvbyten`

A private implementation of the `SvPVbyte` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

char* `sv_pvbyten(SV *sv, STRLEN *len)`

=for hackers Found in file sv.c

`sv_pvbyten_force`

A private implementation of the `SvPVbytex_force` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

char* `sv_pvbyten_force(SV* sv, STRLEN* lp)`

=for hackers Found in file sv.c

`sv_pvn` A private implementation of the `SvPV` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

char* `sv_pvn(SV *sv, STRLEN *len)`

=for hackers Found in file sv.c

`sv_pvn_force`

Get a sensible string out of the SV somehow. A private implementation of the `SvPV_force` macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

char* `sv_pvn_force(SV* sv, STRLEN* lp)`

=for hackers Found in file sv.c

sv_pvn_force_flags

Get a sensible string out of the SV somehow. If flags has SV_GMAGIC bit set, will mg_get on sv if appropriate, else not. sv_pvn_force and sv_pvn_force_nomg are implemented in terms of this function. You normally want to use the various wrapper macros instead: see SvPV_force and SvPV_force_nomg

```
char*    sv_pvn_force_flags(SV* sv, STRLEN* lp, I32 flags)
```

=for hackers Found in file sv.c

sv_pvutf8 Use the SvPVutf8_nolen macro instead

```
char*    sv_pvutf8(SV *sv)
```

=for hackers Found in file sv.c

sv_pvutf8n

A private implementation of the SvPVutf8 macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
char*    sv_pvutf8n(SV *sv, STRLEN *len)
```

=for hackers Found in file sv.c

sv_pvutf8n_force

A private implementation of the SvPVutf8_force macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
char*    sv_pvutf8n_force(SV* sv, STRLEN* lp)
```

=for hackers Found in file sv.c

sv_reftype

Returns a string describing what the SV is a reference to.

```
char*    sv_reftype(SV* sv, int ob)
```

=for hackers Found in file sv.c

sv_replace

Make the first argument a copy of the second, then delete the original. The target SV physically takes over ownership of the body of the source SV and inherits its flags; however, the target keeps any magic it owns, and any magic in the source is discarded. Note that this is a rather specialist SV copying operation; most of the time you'll want to use sv_setsv or one of its many macro front-ends.

```
void     sv_replace(SV* sv, SV* nsv)
```

=for hackers Found in file sv.c

sv_report_used

Dump the contents of all SVs not yet freed. (Debugging aid).

```
void     sv_report_used()
```

=for hackers Found in file sv.c

sv_reset Underlying implementation for the reset Perl function. Note that the perl-level function is vaguely deprecated.

```
void     sv_reset(char* s, HV* stash)
```

=for hackers Found in file sv.c

sv_rvweaken

Weaken a reference: set the SvWEAKREF flag on this RV; give the referred-to SV PERL_MAGIC_backref magic if it hasn't already; and push a back-reference to this RV onto the array of backreferences associated with that magic.

```
SV*      sv_rvweaken(SV *sv)
```

=for hackers Found in file sv.c

sv_setiv Copies an integer into the given SV, upgrading first if necessary. Does not handle 'set' magic. See also sv_setiv_mg.

```
void      sv_setiv(SV* sv, IV num)
```

=for hackers Found in file sv.c

sv_setiv_mg

Like sv_setiv, but also handles 'set' magic.

```
void      sv_setiv_mg(SV *sv, IV i)
```

=for hackers Found in file sv.c

sv_setnv Copies a double into the given SV, upgrading first if necessary. Does not handle 'set' magic. See also sv_setnv_mg.

```
void      sv_setnv(SV* sv, NV num)
```

=for hackers Found in file sv.c

sv_setnv_mg

Like sv_setnv, but also handles 'set' magic.

```
void      sv_setnv_mg(SV *sv, NV num)
```

=for hackers Found in file sv.c

sv_setpv Copies a string into an SV. The string must be null-terminated. Does not handle 'set' magic. See sv_setpv_mg.

```
void      sv_setpv(SV* sv, const char* ptr)
```

=for hackers Found in file sv.c

sv_setpvf Processes its arguments like sprintf and sets an SV to the formatted output. Does not handle 'set' magic. See sv_setpvf_mg.

```
void      sv_setpvf(SV* sv, const char* pat, ...)
```

=for hackers Found in file sv.c

sv_setpvf_mg

Like sv_setpvf, but also handles 'set' magic.

```
void      sv_setpvf_mg(SV *sv, const char* pat, ...)
```

=for hackers Found in file sv.c

sv_setpvn

Copies a string into an SV. The len parameter indicates the number of bytes to be copied. Does not handle 'set' magic. See sv_setpvn_mg.

```
void      sv_setpvn(SV* sv, const char* ptr, STRLEN len)
```

=for hackers Found in file sv.c

sv_setpvn_mg

Like `sv_setpvn`, but also handles ‘set’ magic.

```
void      sv_setpvn_mg(SV *sv, const char *ptr, STRLEN len)
```

=for hackers Found in file `sv.c`

sv_setpv_mg

Like `sv_setpv`, but also handles ‘set’ magic.

```
void      sv_setpv_mg(SV *sv, const char *ptr)
```

=for hackers Found in file `sv.c`

sv_setref_iv

Copies an integer into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*       sv_setref_iv(SV* rv, const char* classname, IV iv)
```

=for hackers Found in file `sv.c`

sv_setref_nv

Copies a double into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*       sv_setref_nv(SV* rv, const char* classname, NV nv)
```

=for hackers Found in file `sv.c`

sv_setref_pvn

Copies a pointer into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. If the `pv` argument is `NULL` then `PL_sv_undef` will be placed into the SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

Do not use with other Perl types such as HV, AV, SV, CV, because those objects will become corrupted by the pointer copy process.

Note that `sv_setref_pvn` copies the string while this copies the pointer.

```
SV*       sv_setref_pvn(SV* rv, const char* classname, void* pv)
```

=for hackers Found in file `sv.c`

sv_setref_pvn

Copies a string into a new SV, optionally blessing the SV. The length of the string must be specified with `n`. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

Note that `sv_setref_pvn` copies the pointer while this copies the string.

```
SV*       sv_setref_pvn(SV* rv, const char* classname, char* pv, STRLEN
```

=for hackers Found in file `sv.c`

sv_setref_uv

Copies an unsigned integer into a new SV, optionally blessing the SV. The `rv` argument will be upgraded to an RV. That RV will be modified to point to the new SV. The `classname` argument indicates the package for the blessing. Set `classname` to `Nullch` to avoid the blessing. The new SV will be returned and will have a reference count of 1.

```
SV*      sv_setref_uv(SV* rv, const char* classname, UV uv)
```

=for hackers Found in file `sv.c`

sv_setsv Copies the contents of the source SV `ssv` into the destination SV `dsv`. The source SV may be destroyed if it is mortal, so don't use this function if the source SV needs to be reused. Does not handle 'set' magic. Loosely speaking, it performs a copy-by-value, obliterating any previous content of the destination.

You probably want to use one of the assortment of wrappers, such as `SvSetSV`, `SvSetSV_nosteal`, `SvSetMagicSV` and `SvSetMagicSV_nosteal`.

```
void      sv_setsv(SV* dsv, SV* ssv)
```

=for hackers Found in file `sv.c`

sv_setsv_flags

Copies the contents of the source SV `ssv` into the destination SV `dsv`. The source SV may be destroyed if it is mortal, so don't use this function if the source SV needs to be reused. Does not handle 'set' magic. Loosely speaking, it performs a copy-by-value, obliterating any previous content of the destination. If the `flags` parameter has the `SV_GMAGIC` bit set, will `mg_get` on `ssv` if appropriate, else not. `sv_setsv` and `sv_setsv_nomg` are implemented in terms of this function.

You probably want to use one of the assortment of wrappers, such as `SvSetSV`, `SvSetSV_nosteal`, `SvSetMagicSV` and `SvSetMagicSV_nosteal`.

This is the primary function for copying scalars, and most other copy-ish functions and macros use this underneath.

```
void      sv_setsv_flags(SV* dsv, SV* ssv, I32 flags)
```

=for hackers Found in file `sv.c`

sv_setsv_mg

Like `sv_setsv`, but also handles 'set' magic.

```
void      sv_setsv_mg(SV *dstr, SV *sstr)
```

=for hackers Found in file `sv.c`

sv_setuv Copies an unsigned integer into the given SV, upgrading first if necessary. Does not handle 'set' magic. See also `sv_setuv_mg`.

```
void      sv_setuv(SV* sv, UV num)
```

=for hackers Found in file `sv.c`

sv_setuv_mg

Like `sv_setuv`, but also handles 'set' magic.

```
void      sv_setuv_mg(SV *sv, UV u)
```

=for hackers Found in file `sv.c`

sv_taint Taint an SV. Use `SvTAINTED_on` instead.

```
void      sv_taint(SV* sv)
```


=for hackers Found in file sv.c

sv_tainted

Test an SV for taintedness. Use SvTAINTED instead.

```
bool      sv_tainted(SV* sv)
```

=for hackers Found in file sv.c

sv_true Returns true if the SV has a true value by Perl's rules. Use the SvTRUE macro instead, which may call sv_true() or may instead use an in-line version.

```
I32      sv_true(SV *sv)
```

=for hackers Found in file sv.c

sv_unmagic

Removes all magic of type `type` from an SV.

```
int      sv_unmagic(SV* sv, int type)
```

=for hackers Found in file sv.c

sv_unref Unsets the RV status of the SV, and decrements the reference count of whatever was being referenced by the RV. This can almost be thought of as a reversal of newSVrv. This is sv_unref_flags with the flag being zero. See SvROK_off.

```
void     sv_unref(SV* sv)
```

=for hackers Found in file sv.c

sv_unref_flags

Unsets the RV status of the SV, and decrements the reference count of whatever was being referenced by the RV. This can almost be thought of as a reversal of newSVrv. The cflags argument can contain SV_IMMEDIATE_UNREF to force the reference count to be decremented (otherwise the decrementing is conditional on the reference count being different from one or the reference being a readonly SV). See SvROK_off.

```
void     sv_unref_flags(SV* sv, U32 flags)
```

=for hackers Found in file sv.c

sv_untaint

Untaint an SV. Use SvTAINTED_off instead.

```
void     sv_untaint(SV* sv)
```

=for hackers Found in file sv.c

sv_upgrade

Upgrade an SV to a more complex form. Generally adds a new body type to the SV, then copies across as much information as possible from the old body. You generally want to use the SvUPGRADE macro wrapper. See also svtype.

```
bool     sv_upgrade(SV* sv, U32 mt)
```

=for hackers Found in file sv.c

sv_usepvn

Tells an SV to use ptr to find its string value. Normally the string is stored inside the SV but sv_usepvn allows the SV to use an outside string. The ptr should point to memory that was allocated by malloc. The string length, len, must be supplied. This function will realloc the memory pointed to by ptr, so that pointer should not be freed or used by the programmer after giving it to sv_usepvn. Does not handle 'set' magic. See sv_usepvn_mg.

```
void    sv_usepvn(SV* sv, char* ptr, STRLEN len)
```

=for hackers Found in file sv.c

sv_usepvn_mg

Like `sv_usepvn`, but also handles ‘set’ magic.

```
void    sv_usepvn_mg(SV *sv, char *ptr, STRLEN len)
```

=for hackers Found in file sv.c

sv_utf8_decode

Convert the octets in the PV from UTF-8 to chars. Scan for validity and then turn off SvUTF8 if needed so that we see characters. Used as a building block for `decode_utf8` in `Encode.xs`

NOTE: this function is experimental and may change or be removed without notice.

```
bool    sv_utf8_decode(SV *sv)
```

=for hackers Found in file sv.c

sv_utf8_downgrade

Attempt to convert the PV of an SV from UTF8-encoded to byte encoding. This may not be possible if the PV contains non-byte encoding characters; if this is the case, either returns false or, if `fail_ok` is not true, croaks.

This is not as a general purpose Unicode to byte encoding interface: use the `Encode` extension for that.

NOTE: this function is experimental and may change or be removed without notice.

```
bool    sv_utf8_downgrade(SV *sv, bool fail_ok)
```

=for hackers Found in file sv.c

sv_utf8_encode

Convert the PV of an SV to UTF8-encoded, but then turn off the SvUTF8 flag so that it looks like octets again. Used as a building block for `encode_utf8` in `Encode.xs`

```
void    sv_utf8_encode(SV *sv)
```

=for hackers Found in file sv.c

sv_utf8_upgrade

Convert the PV of an SV to its UTF8-encoded form. Forces the SV to string form if it is not already. Always sets the SvUTF8 flag to avoid future validity checks even if all the bytes have hibit clear.

This is not as a general purpose byte encoding to Unicode interface: use the `Encode` extension for that.

```
STRLEN  sv_utf8_upgrade(SV *sv)
```

=for hackers Found in file sv.c

sv_utf8_upgrade_flags

Convert the PV of an SV to its UTF8-encoded form. Forces the SV to string form if it is not already. Always sets the SvUTF8 flag to avoid future validity checks even if all the bytes have hibit clear. If `flags` has `SV_GMAGIC` bit set, will `mg_get` on `sv` if appropriate, else not. `sv_utf8_upgrade` and `sv_utf8_upgrade_nomg` are implemented in terms of this function.

This is not as a general purpose byte encoding to Unicode interface: use the `Encode` extension for that.

```
STRLEN sv_utf8_upgrade_flags(SV *sv, I32 flags)
```

=for hackers Found in file sv.c

sv_uv A private implementation of the SvUVx macro for compilers which can't cope with complex macro expressions. Always use the macro instead.

```
UV sv_uv(SV* sv)
```

=for hackers Found in file sv.c

sv_vcatpvfn

Processes its arguments like vsprintf and appends the formatted output to an SV. Uses an array of SVs if the C style variable argument list is missing (NULL). When running with taint checks enabled, indicates via maybe_tainted if results are untrustworthy (often due to the use of locales).

Usually used via one of its frontends sv_catpvf and sv_catpvf_mg.

```
void sv_vcatpvfn(SV* sv, const char* pat, STRLEN patlen, va_list*
```

=for hackers Found in file sv.c

sv_vsetpvfn

Works like vcatpvfn but copies the text into the SV instead of appending it.

Usually used via one of its frontends sv_setpvf and sv_setpvf_mg.

```
void sv_vsetpvfn(SV* sv, const char* pat, STRLEN patlen, va_list*
```

=for hackers Found in file sv.c

Unicode Support

bytes_from_utf8

Converts a string *s* of length *len* from UTF8 into byte encoding. Unlike <utf8_to_bytes but like bytes_to_utf8, returns a pointer to the newly-created string, and updates *len* to contain the new length. Returns the original string if no conversion occurs, *len* is unchanged. Do nothing if *is_utf8* points to 0. Sets *is_utf8* to 0 if *s* is converted or contains all 7bit characters.

NOTE: this function is experimental and may change or be removed without notice.

```
U8* bytes_from_utf8(U8 *s, STRLEN *len, bool *is_utf8)
```

=for hackers Found in file utf8.c

bytes_to_utf8

Converts a string *s* of length *len* from ASCII into UTF8 encoding. Returns a pointer to the newly-created string, and sets *len* to reflect the new length.

NOTE: this function is experimental and may change or be removed without notice.

```
U8* bytes_to_utf8(U8 *s, STRLEN *len)
```

=for hackers Found in file utf8.c

ibcmp_utf8

Return true if the strings *s1* and *s2* differ case-insensitively, false if not (if they are equal case-insensitively). If *u1* is true, the string *s1* is assumed to be in UTF-8-encoded Unicode. If *u2* is true, the string *s2* is assumed to be in UTF-8-encoded Unicode. If *u1* or *u2* are false, the respective string is assumed to be in native 8-bit encoding.

If the *pe1* and *pe2* are non-NULL, the scanning pointers will be copied in there (they will point at the beginning of the *next* character). If the pointers behind *pe1* or *pe2* are non-NULL, they are

the end pointers beyond which scanning will not continue under any circumstances. If the byte lengths l1 and l2 are non-zero, s1+l1 and s2+l2 will be used as goal end pointers that will also stop the scan, and which qualify towards defining a successful match: all the scans that define an explicit length must reach their goal pointers for a match to succeed).

For case-insensitiveness, the "casefolding" of Unicode is used instead of upper/lowercasing both the characters, see <http://www.unicode.org/unicode/reports/tr21/> (Case Mappings).

```
I32      ibcmp_utf8(const char* a, char **pe1, UV l1, bool u1, const c
```

=for hackers Found in file utf8.c

is_utf8_char

Tests if some arbitrary number of bytes begins in a valid UTF-8 character. Note that an INVARIANT (i.e. ASCII) character is a valid UTF-8 character. The actual number of bytes in the UTF-8 character will be returned if it is valid, otherwise 0.

```
STRLEN  is_utf8_char(U8 *p)
```

=for hackers Found in file utf8.c

is_utf8_string

Returns true if first len bytes of the given string form a valid UTF8 string, false otherwise. Note that 'a valid UTF8 string' does not mean 'a string that contains UTF8' because a valid ASCII string is a valid UTF8 string.

```
bool     is_utf8_string(U8 *s, STRLEN len)
```

=for hackers Found in file utf8.c

pv_uni_display

Build to the scalar dsv a displayable version of the string spv, length len, the displayable version being at most pvlm bytes long (if longer, the rest is truncated and "..." will be appended).

The flags argument can have UNI_DISPLAY_ISPRINT set to display isPRINT()able characters as themselves, UNI_DISPLAY_BACKSLASH to display the `\\[nrfta\\]` as the backslashed versions (like `'\n'`) (UNI_DISPLAY_BACKSLASH is preferred over UNI_DISPLAY_ISPRINT for `\\`). UNI_DISPLAY_QQ (and its alias UNI_DISPLAY_REGEX) have both UNI_DISPLAY_BACKSLASH and UNI_DISPLAY_ISPRINT turned on.

The pointer to the PV of the dsv is returned.

```
char*    pv_uni_display(SV *dsv, U8 *spv, STRLEN len, STRLEN pvlm, UV
```

=for hackers Found in file utf8.c

sv_recode_to_utf8

The encoding is assumed to be an Encode object, on entry the PV of the sv is assumed to be octets in that encoding, and the sv will be converted into Unicode (and UTF-8).

If the sv already is UTF-8 (or if it is not POK), or if the encoding is not a reference, nothing is done to the sv. If the encoding is not an Encode : XS Encoding object, bad things will happen. (See *lib/encoding.pm* and *Encode*).

The PV of the sv is returned.

```
char*    sv_recode_to_utf8(SV* sv, SV *encoding)
```

=for hackers Found in file sv.c

sv_uni_display

Build to the scalar dsv a displayable version of the scalar sv, the displayable version being at most pvlm bytes long (if longer, the rest is truncated and "..." will be appended).

The flags argument is as in `pv_uni_display()`.

The pointer to the PV of the dsv is returned.

```
char*    sv_uni_display(SV *dsv, SV *ssv, STRLEN pvlm, UV flags)
```

=for hackers Found in file utf8.c

to_utf8_case

The "p" contains the pointer to the UTF-8 string encoding the character that is being converted.

The "ustrp" is a pointer to the character buffer to put the conversion result to. The "lenp" is a pointer to the length of the result.

The "swashp" is a pointer to the swash to use.

Both the special and normal mappings are stored lib/unicore/To/Foo.pl, and loaded by SWASHGET, using lib/utf8_heavy.pl. The special (usually, but not always, a multicharacter mapping), is tried first.

The "special" is a string like "utf8::ToSpecLower", which means the hash %utf8::ToSpecLower. The access to the hash is through `Perl_to_utf8_case()`.

The "normal" is a string like "ToLower" which means the swash %utf8::ToLower.

```
UV        to_utf8_case(U8 *p, U8* ustrp, STRLEN *lenp, SV **swash, char
```

=for hackers Found in file utf8.c

to_utf8_fold

Convert the UTF-8 encoded character at p to its foldcase version and store that in UTF-8 in ustrp and its length in bytes in lenp. Note that the ustrp needs to be at least UTF8_MAXLEN_FOLD+1 bytes since the foldcase version may be longer than the original character (up to three characters).

The first character of the foldcased version is returned (but note, as explained above, that there may be more.)

```
UV        to_utf8_fold(U8 *p, U8* ustrp, STRLEN *lenp)
```

=for hackers Found in file utf8.c

to_utf8_lower

Convert the UTF-8 encoded character at p to its lowercase version and store that in UTF-8 in ustrp and its length in bytes in lenp. Note that the ustrp needs to be at least UTF8_MAXLEN_UCLC+1 bytes since the lowercase version may be longer than the original character (up to two characters).

The first character of the lowercased version is returned (but note, as explained above, that there may be more.)

```
UV        to_utf8_lower(U8 *p, U8* ustrp, STRLEN *lenp)
```

=for hackers Found in file utf8.c

to_utf8_title

Convert the UTF-8 encoded character at p to its titlecase version and store that in UTF-8 in ustrp and its length in bytes in lenp. Note that the ustrp needs to be at least UTF8_MAXLEN_UCLC+1 bytes since the titlecase version may be longer than the original character (up to two characters).

The first character of the titlecased version is returned (but note, as explained above, that there may be more.)

```
UV      to_utf8_title(U8 *p, U8* ustrp, STRLEN *lenp)
```

=for hackers Found in file utf8.c

to_utf8_upper

Convert the UTF-8 encoded character at *p* to its uppercase version and store that in UTF-8 in *ustrp* and its length in bytes in *lenp*. Note that the *ustrp* needs to be at least UTF8_MAXLEN_UCLC+1 bytes since the uppercase version may be longer than the original character (up to two characters).

The first character of the uppercased version is returned (but note, as explained above, that there may be more.)

```
UV      to_utf8_upper(U8 *p, U8* ustrp, STRLEN *lenp)
```

=for hackers Found in file utf8.c

utf8n_to_uvchr

Returns the native character value of the first character in the string *s* which is assumed to be in UTF8 encoding; *retlen* will be set to the length, in bytes, of that character.

Allows length and flags to be passed to low level routine.

```
UV      utf8n_to_uvchr(U8 *s, STRLEN curlen, STRLEN* retlen, U32 flag)
```

=for hackers Found in file utf8.c

utf8n_to_uvuni

Bottom level UTF-8 decode routine. Returns the unicode code point value of the first character in the string *s* which is assumed to be in UTF8 encoding and no longer than *curlen*; *retlen* will be set to the length, in bytes, of that character.

If *s* does not point to a well-formed UTF8 character, the behaviour is dependent on the value of *flags*: if it contains UTF8_CHECK_ONLY, it is assumed that the caller will raise a warning, and this function will silently just set *retlen* to -1 and return zero. If the *flags* does not contain UTF8_CHECK_ONLY, warnings about malformations will be given, *retlen* will be set to the expected length of the UTF-8 character in bytes, and zero will be returned.

The *flags* can also contain various flags to allow deviations from the strict UTF-8 encoding (see *utf8.h*).

Most code should use `utf8_to_uvchr()` rather than call this directly.

```
UV      utf8n_to_uvuni(U8 *s, STRLEN curlen, STRLEN* retlen, U32 flag)
```

=for hackers Found in file utf8.c

utf8_distance

Returns the number of UTF8 characters between the UTF-8 pointers *a* and *b*.

WARNING: use only if you **know** that the pointers point inside the same UTF-8 buffer.

```
IV      utf8_distance(U8 *a, U8 *b)
```

=for hackers Found in file utf8.c

utf8_hop

Return the UTF-8 pointer *s* displaced by *off* characters, either forward or backward.

WARNING: do not use the following unless you **know** *off* is within the UTF-8 data pointed to by *s* **and** that on entry *s* is aligned on the first byte of character or just after the last byte of a character.

```
U8*     utf8_hop(U8 *s, I32 off)
```

=for hackers Found in file utf8.c

utf8_length

Return the length of the UTF-8 char encoded string *s* in characters. Stops at *e* (inclusive). If *e* < *s* or if the scan would end up past *e*, croaks.

```
STRLEN utf8_length(U8* s, U8 *e)
```

=for hackers Found in file utf8.c

utf8_to_bytes

Converts a string *s* of length *len* from UTF8 into byte encoding. Unlike `bytes_to_utf8`, this over-writes the original string, and updates *len* to contain the new length. Returns zero on failure, setting *len* to -1.

NOTE: this function is experimental and may change or be removed without notice.

```
U8* utf8_to_bytes(U8 *s, STRLEN *len)
```

=for hackers Found in file utf8.c

utf8_to_uvchr

Returns the native character value of the first character in the string *s* which is assumed to be in UTF8 encoding; *retlen* will be set to the length, in bytes, of that character.

If *s* does not point to a well-formed UTF8 character, zero is returned and *retlen* is set, if possible, to -1.

```
UV utf8_to_uvchr(U8 *s, STRLEN* retlen)
```

=for hackers Found in file utf8.c

utf8_to_uvuni

Returns the Unicode code point of the first character in the string *s* which is assumed to be in UTF8 encoding; *retlen* will be set to the length, in bytes, of that character.

This function should only be used when returned UV is considered an index into the Unicode semantic tables (e.g. swashes).

If *s* does not point to a well-formed UTF8 character, zero is returned and *retlen* is set, if possible, to -1.

```
UV utf8_to_uvuni(U8 *s, STRLEN* retlen)
```

=for hackers Found in file utf8.c

uvchr_to_utf8

Adds the UTF8 representation of the Native codepoint *uv* to the end of the string *d*; *d* should be have at least `UTF8_MAXLEN+1` free bytes available. The return value is the pointer to the byte after the end of the new character. In other words,

```
d = uvchr_to_utf8(d, uv);
```

is the recommended wide native character-aware way of saying

```
*(d++) = uv;
```

```
U8* uvchr_to_utf8(U8 *d, UV uv)
```

=for hackers Found in file utf8.c

uvuni_to_utf8_flags

Adds the UTF8 representation of the Unicode codepoint *uv* to the end of the string *d*; *d* should be have at least `UTF8_MAXLEN+1` free bytes available. The return value is the pointer to the byte after the end of the new character. In other words,

```

        d = uvuni_to_utf8_flags(d, uv, flags);
or, in most cases,
        d = uvuni_to_utf8(d, uv);
(which is equivalent to)
        d = uvuni_to_utf8_flags(d, uv, 0);
is the recommended Unicode-aware way of saying
        *(d++) = uv;
        U8*      uvuni_to_utf8_flags(U8 *d, UV uv, UV flags)
=for hackers Found in file utf8.c

```

Variables created by xsubpp and xsubpp internal functions

ax Variable which is setup by xsubpp to indicate the stack base offset, used by the ST, XSpREPush and XSRETURN macros. The dMARK macro must be called prior to setup the MARK variable.

```
I32      ax
```

=for hackers Found in file XSUB.h

CLASS Variable which is setup by xsubpp to indicate the class name for a C++ XS constructor. This is always a char*. See THIS.

```
char*    CLASS
```

=for hackers Found in file XSUB.h

dAX Sets up the ax variable. This is usually handled automatically by xsubpp by calling dXSARGS.

```
dAX;
```

=for hackers Found in file XSUB.h

dITEMS Sets up the items variable. This is usually handled automatically by xsubpp by calling dXSARGS.

```
dITEMS;
```

=for hackers Found in file XSUB.h

dXSARGS

Sets up stack and mark pointers for an XSUB, calling dSP and dMARK. Sets up the ax and items variables by calling dAX and dITEMS. This is usually handled automatically by xsubpp.

```
dXSARGS;
```

=for hackers Found in file XSUB.h

dXS132 Sets up the ix variable for an XSUB which has aliases. This is usually handled automatically by xsubpp.

```
dXS132;
```

=for hackers Found in file XSUB.h

items Variable which is setup by xsubpp to indicate the number of items on the stack. See [Variable-length Parameter Lists in perlxs](#).

- `I32` `items`
 =for hackers Found in file XSUB.h
- ix** Variable which is setup by `xsubpp` to indicate which of an `XSUB`'s aliases was used to invoke it. See [The *ALIAS*: Keyword in *perlxs*](#).
- `I32` `ix`
 =for hackers Found in file XSUB.h
- newXSproto**
 Used by `xsubpp` to hook up `XSUB`s as Perl subs. Adds Perl prototypes to the subs.
 =for hackers Found in file XSUB.h
- RETVAL** Variable which is setup by `xsubpp` to hold the return value for an `XSUB`. This is always the proper type for the `XSUB`. See [The *RETVAL* Variable in *perlxs*](#).
- `(whatever)` `RETVAL`
 =for hackers Found in file XSUB.h
- ST** Used to access elements on the `XSUB`'s stack.
- `SV*` `ST(int ix)`
 =for hackers Found in file XSUB.h
- THIS** Variable which is setup by `xsubpp` to designate the object in a C++ `XSUB`. This is always the proper type for the C++ object. See `CLASS` and [Using *XS* With C++ in *perlxs*](#).
- `(whatever)` `THIS`
 =for hackers Found in file XSUB.h
- XS** Macro to declare an `XSUB` and its C parameter list. This is handled by `xsubpp`.
 =for hackers Found in file XSUB.h
- XSRETURN_EMPTY**
 Return an empty list from an `XSUB` immediately.
- `XSRETURN_EMPTY;`
 =for hackers Found in file XSUB.h
- XS_VERSION**
 The version identifier for an `XS` module. This is usually handled automatically by `ExtUtils::MakeMaker`. See `XS_VERSION_BOOTCHECK`.
 =for hackers Found in file XSUB.h
- XS_VERSION_BOOTCHECK**
 Macro to verify that a PM module's `$VERSION` variable matches the `XS` module's `XS_VERSION` variable. This is usually handled automatically by `xsubpp`. See [The *VERSIONCHECK*: Keyword in *perlxs*](#).
- `XS_VERSION_BOOTCHECK;`
 =for hackers Found in file XSUB.h

Warning and Dieing

- croak** This is the `XSUB`-writer's interface to Perl's `die` function. Normally use this function the same way you use the C `printf` function. See `warn`.

If you want to throw an exception object, assign the object to `$@` and then pass `Nullch` to `croak()`:

```
errsv = get_sv("@", TRUE);
sv_setsv(errsv, exception_object);
croak(Nullch);

void    croak(const char* pat, ...)
```

=for hackers Found in file util.c

warn This is the XSUB-writer's interface to Perl's `warn` function. Use this function the same way you use the C `printf` function. See `croak`.

```
void    warn(const char* pat, ...)
```

=for hackers Found in file util.c

AUTHORS

Until May 1997, this document was maintained by Jeff Okamoto <okamoto@corp.hp.com>. It is now maintained as part of Perl itself.

With lots of help and suggestions from Dean Roehrich, Malcolm Beattie, Andreas Koenig, Paul Hudson, Ilya Zakharevich, Paul Marquess, Neil Bowers, Matthew Green, Tim Bunce, Spider Boardman, Ulrich Pfeifer, Stephen McCamant, and Gurusamy Sarathy.

API Listing originally by Dean Roehrich <roehrich@cray.com>.

Updated to be autogenerated from comments in the source by Benjamin Stuhl.

SEE ALSO

`perlguits(1)`, `perlxs(1)`, `perlxstut(1)`, `perlintern(1)`

NAME

perlpio – perl's IO abstraction interface.

SYNOPSIS

```
#define PERLIO_NOT_STDIO 0      /* For co-existence with stdio only */
#include <perl.h>              /* Usually via #include <perl.h> */

PerlIO *PerlIO_stdin(void);
PerlIO *PerlIO_stdout(void);
PerlIO *PerlIO_stderr(void);

PerlIO *PerlIO_open(const char *path, const char *mode);
PerlIO *PerlIO_fdopen(int fd, const char *mode);
PerlIO *PerlIO_reopen(const char *path, const char *mode, PerlIO *old); /* deprecated */
int PerlIO_close(PerlIO *f);

int PerlIO_stdoutf(const char *fmt, ...)
int PerlIO_puts(PerlIO *f, const char *string);
int PerlIO_putc(PerlIO *f, int ch);
int PerlIO_write(PerlIO *f, const void *buf, size_t numbytes);
int PerlIO_printf(PerlIO *f, const char *fmt, ...);
int PerlIO_vprintf(PerlIO *f, const char *fmt, va_list args);
int PerlIO_flush(PerlIO *f);

int PerlIO_eof(PerlIO *f);
int PerlIO_error(PerlIO *f);
void PerlIO_clearerr(PerlIO *f);

int PerlIO_getc(PerlIO *d);
int PerlIO_ungetc(PerlIO *f, int ch);
int PerlIO_read(PerlIO *f, void *buf, size_t numbytes);

int PerlIO_fileno(PerlIO *f);

void PerlIO_setlinebuf(PerlIO *f);

Off_t PerlIO_tell(PerlIO *f);
int PerlIO_seek(PerlIO *f, Off_t offset, int whence);
void PerlIO_rewind(PerlIO *f);

int PerlIO_getpos(PerlIO *f, SV *save); /* prototype changed */
int PerlIO_setpos(PerlIO *f, SV *saved); /* prototype changed */

int PerlIO_fast_gets(PerlIO *f);
int PerlIO_has_cntptr(PerlIO *f);
int PerlIO_get_cnt(PerlIO *f);
char *PerlIO_get_ptr(PerlIO *f);
void PerlIO_set_ptrcnt(PerlIO *f, char *ptr, int count);

int PerlIO_canset_cnt(PerlIO *f); /* deprecated */
void PerlIO_set_cnt(PerlIO *f, int count); /* deprecated */

int PerlIO_has_base(PerlIO *f);
char *PerlIO_get_base(PerlIO *f);
int PerlIO_get_bufsiz(PerlIO *f);

PerlIO *PerlIO_importFILE(FILE *stdio, const char *mode);
FILE *PerlIO_exportFILE(PerlIO *f, int flags);
FILE *PerlIO_findFILE(PerlIO *f);
void PerlIO_releaseFILE(PerlIO *f, FILE *stdio);
```

```

int      PerlIO_apply_layers(PerlIO *f, const char *mode, const char *layers);
int      PerlIO_binmode(PerlIO *f, int ptype, int imode, const char *layers);
void     PerlIO_debug(const char *fmt,...)

```

DESCRIPTION

Perl's source code, and extensions that want maximum portability, should use the above functions instead of those defined in ANSI C's *stdio.h*. The perl headers (in particular "perlio.h") will #define them to the I/O mechanism selected at Configure time.

The functions are modeled on those in *stdio.h*, but parameter order has been "tidied up a little".

PerlIO * takes the place of FILE *. Like FILE * it should be treated as opaque (it is probably safe to assume it is a pointer to something).

There are currently three implementations:

1. USE_STDIO

All above are #define'd to stdio functions or are trivial wrapper functions which call stdio. In this case *only* PerlIO * is a FILE *. This has been the default implementation since the abstraction was introduced in perl5.003_02.

2. USE_SFIO

A "legacy" implementation in terms of the "sfio" library. Used for some specialist applications on Unix machines ("sfio" is not widely ported away from Unix). Most of above are #define'd to the sfio functions. PerlIO * is in this case Sfile_t *.

3. USE_PERLIO

Introduced just after perl5.7.0, this is a re-implementation of the above abstraction which allows perl more control over how IO is done as it decouples IO from the way the operating system and C library choose to do things. For USE_PERLIO PerlIO * has an extra layer of indirection – it is a pointer-to-a-pointer. This allows the PerlIO * to remain with a known value while swapping the implementation around underneath *at run time*. In this case all the above are true (but very simple) functions which call the underlying implementation.

This is the only implementation for which PerlIO_apply_layers() does anything "interesting".

The USE_PERLIO implementation is described in [perliol](#).

Because "perlio.h" is a thin layer (for efficiency) the semantics of these functions are somewhat dependent on the underlying implementation. Where these variations are understood they are noted below.

Unless otherwise noted, functions return 0 on success, or a negative value (usually EOF which is usually -1) and set errno on error.

PerlIO_stdin(), PerlIO_stdout(), PerlIO_stderr()

Use these rather than stdin, stdout, stderr. They are written to look like "function calls" rather than variables because this makes it easier to *make them* function calls if platform cannot export data to loaded modules, or if (say) different "threads" might have different values.

PerlIO_open(path, mode), PerlIO_fdopen(fd,mode)

These correspond to fopen()/fdopen() and the arguments are the same. Return NULL and set errno if there is an error. There may be an implementation limit on the number of open handles, which may be lower than the limit on the number of open files – errno may not be set when NULL is returned if this limit is exceeded.

PerlIO_reopen(path,mode,f)

While this currently exists in all three implementations perl itself does not use it. *As perl does not use it, it is not well tested.*

Perl prefers to dup the new low-level descriptor to the descriptor used by the existing PerlIO. This may become the behaviour of this function in the future.

PerlIO_printf(f,fmt,...), PerlIO_vprintf(f,fmt,a)

These are `fprintf()` / `vfprintf()` equivalents.

PerlIO_stdoutf(fmt,...)

This is `printf()` equivalent. `printf` is #defined to this function, so it is (currently) legal to use `printf(fmt,...)` in perl sources.

PerlIO_read(f,buf,count), PerlIO_write(f,buf,count)

These correspond to `fread()` and `fwrite()`. Note that arguments are different, there is only one "count" and order has "file" first. Returns a byte count if successful (which may be zero or positive), returns negative value and sets `errno` on error. Depending on implementation `errno` may be `EINTR` if operation was interrupted by a signal.

PerlIO_close(f)

Depending on implementation `errno` may be `EINTR` if operation was interrupted by a signal.

PerlIO_puts(f,s), PerlIO_putc(f,c)

These correspond to `fputs()` and `fputc()`. Note that arguments have been revised to have "file" first.

PerlIO_ungetc(f,c)

This corresponds to `ungetc()`. Note that arguments have been revised to have "file" first. Arranges that next read operation will return the byte `c`. Despite the implied "character" in the name only values in the range `0..0xFF` are defined. Returns the byte `c` on success or `-1` (EOF) on error. The number of bytes that can be "pushed back" may vary, only 1 character is certain, and then only if it is the last character that was read from the handle.

PerlIO_getc(f)

This corresponds to `getc()`. Despite the `c` in the name only byte range `0..0xFF` is supported. Returns the character read or `-1` (EOF) on error.

PerlIO_eof(f)

This corresponds to `feof()`. Returns a true/false indication of whether the handle is at end of file. For terminal devices this may or may not be "sticky" depending on the implementation. The flag is cleared by `PerlIO_seek()`, or `PerlIO_rewind()`.

PerlIO_error(f)

This corresponds to `ferror()`. Returns a true/false indication of whether there has been an IO error on the handle.

PerlIO_fileno(f)

This corresponds to `fileno()`, note that on some platforms, the meaning of "fileno" may not match Unix. Returns `-1` if the handle has no open descriptor associated with it.

PerlIO_clearerr(f)

This corresponds to `clearerr()`, i.e., clears 'error' and (usually) 'eof' flags for the "stream". Does not return a value.

PerlIO_flush(f)

This corresponds to `fflush()`. Sends any buffered write data to the underlying file. If called with `NULL` this may flush all open streams (or core dump with some `USE_STDIO` implementations). Calling on a handle open for read only, or on which last operation was a read of some kind may lead to undefined behaviour on some `USE_STDIO` implementations. The `USE_PERLIO` (layers) implementation tries to behave better: it flushes all open streams when passed `NULL`, and attempts to retain data on read streams either in the buffer or by seeking the handle to the current logical position.

PerlIO_seek(f,offset,whence)

This corresponds to `fseek()`. Sends buffered write data to the underlying file, or discards any buffered read data, then positions the file descriptor as specified by **offset** and **whence** (sic). This is the correct thing to do when switching between read and write on the same handle (see issues with `PerlIO_flush()` above). Offset is of type `Off_t` which is a perl Configure value which may not be same as `stdio's off_t`.

PerlIO_tell(f)

This corresponds to `ftell()`. Returns the current file position, or `(Off_t) -1` on error. May just return value system "knows" without making a system call or checking the underlying file descriptor (so use on shared file descriptors is not safe without a `PerlIO_seek()`). Return value is of type `Off_t` which is a perl Configure value which may not be same as `stdio's off_t`.

PerlIO_getpos(f,p), PerlIO_setpos(f,p)

These correspond (loosely) to `fgetpos()` and `fsetpos()`. Rather than `stdio's Fpos_t` they expect a "Perl Scalar Value" to be passed. What is stored there should be considered opaque. The layout of the data may vary from handle to handle. When not using `stdio` or if platform does not have the `stdio` calls then they are implemented in terms of `PerlIO_tell()` and `PerlIO_seek()`.

PerlIO_rewind(f)

This corresponds to `rewind()`. It is usually defined as being

```
PerlIO_seek(f, (Off_t) 0L, SEEK_SET);
PerlIO_clearerr(f);
```

PerlIO_tmpfile()

This corresponds to `tmpfile()`, i.e., returns an anonymous `PerlIO` or `NULL` on error. The system will attempt to automatically delete the file when closed. On Unix the file is usually `unlink`-ed just after it is created so it does not matter how it gets closed. On other systems the file may only be deleted if closed via `PerlIO_close()` and/or the program exits via `exit`. Depending on the implementation there may be "race conditions" which allow other processes access to the file, though in general it will be safer in this regard than ad. hoc. schemes.

PerlIO_setlinebuf(f)

This corresponds to `setlinebuf()`. Does not return a value. What constitutes a "line" is implementation dependent but usually means that writing `"\n"` flushes the buffer. What happens with things like `"this\nthat"` is uncertain. (Perl core uses it *only* when "dumping"; it has nothing to do with `$|` auto-flush.)

Co-existence with stdio

There is outline support for co-existence of `PerlIO` with `stdio`. Obviously if `PerlIO` is implemented in terms of `stdio` there is no problem. However in other cases then mechanisms must exist to create a `FILE *` which can be passed to library code which is going to use `stdio` calls.

The first step is to add this line:

```
#define PERLIO_NOT_STDIO 0
```

before including any perl header files. (This will probably become the default at some point). That prevents "perl.h" from attempting to `#define` `stdio` functions onto `PerlIO` functions.

XS code is probably better using "typemap" if it expects `FILE *` arguments. The standard typemap will be adjusted to comprehend any changes in this area.

PerlIO_importFILE(f,mode)

Used to get a `PerlIO *` from a `FILE *`.

The mode argument should be a string as would be passed to `fopen/PerlIO_open`. If it is `NULL` then – for legacy support – the code will (depending upon the platform and the implementation) either

attempt to empirically determine the mode in which *f* is open, or use "r+" to indicate a read/write stream.

Once called the FILE * should *ONLY* be closed by calling `PerlIO_close()` on the returned PerlIO *.

The PerlIO is set to textmode. Use `PerlIO_binmode` if this is not the desired mode.

This is **not** the reverse of `PerlIO_exportFILE()`.

PerlIO_exportFILE(f,mode)

Given a PerlIO * create a 'native' FILE * suitable for passing to code expecting to be compiled and linked with ANSI C *stdio.h*. The mode argument should be a string as would be passed to `fopen/PerlIO_open`. If it is NULL then – for legacy support – the FILE * is opened in same mode as the PerlIO *.

The fact that such a FILE * has been 'exported' is recorded, (normally by pushing a new :stdio "layer" onto the PerlIO *), which may affect future PerlIO operations on the original PerlIO *. You should not call `fclose()` on the file unless you call `PerlIO_releaseFILE()` to disassociate it from the PerlIO *. (Do not use `PerlIO_importFILE()` for doing the disassociation.)

Calling this function repeatedly will create a FILE * on each call (and will push an :stdio layer each time as well).

PerlIO_releaseFILE(p,f)

Calling `PerlIO_releaseFILE` informs PerlIO that all use of FILE * is complete. It is removed from the list of 'exported' FILE *s, and the associated PerlIO * should revert to its original behaviour.

Use this to disassociate a file from a PerlIO * that was associated using `PerlIO_exportFILE()`.

PerlIO_findFILE(f)

Returns a native FILE * used by a stdio layer. If there is none, it will create one with `PerlIO_exportFILE`. In either case the FILE * should be considered as belonging to PerlIO subsystem and should only be closed by calling `PerlIO_close()`.

"Fast gets" Functions

In addition to standard-like API defined so far above there is an "implementation" interface which allows perl to get at internals of PerlIO. The following calls correspond to the various FILE_xxx macros determined by Configure – or their equivalent in other implementations. This section is really of interest to only those concerned with detailed perl-core behaviour, implementing a PerlIO mapping or writing code which can make use of the "read ahead" that has been done by the IO system in the same way perl does. Note that any code that uses these interfaces must be prepared to do things the traditional way if a handle does not support them.

PerlIO_fast_gets(f)

Returns true if implementation has all the interfaces required to allow perl's `sv_gets` to "bypass" normal IO mechanism. This can vary from handle to handle.

```
PerlIO_fast_gets(f) = PerlIO_has_cntptr(f) && \
                    PerlIO_canset_cnt(f) && \
                    'Can set pointer into buffer'
```

PerlIO_has_cntptr(f)

Implementation can return pointer to current position in the "buffer" and a count of bytes available in the buffer. Do not use this – use `PerlIO_fast_gets`.

PerlIO_get_cnt(f)

Return count of readable bytes in the buffer. Zero or negative return means no more bytes available.

PerlIO_get_ptr(f)

Return pointer to next readable byte in buffer, accessing via the pointer (dereferencing) is only safe if `PerlIO_get_cnt()` has returned a positive value. Only positive offsets up to value returned by `PerlIO_get_cnt()` are allowed.

PerlIO_set_ptrcnt(f,p,c)

Set pointer into buffer, and a count of bytes still in the buffer. Should be used only to set pointer to within range implied by previous calls to `PerlIO_get_ptr` and `PerlIO_get_cnt`. The two values *must* be consistent with each other (implementation may only use one or the other or may require both).

PerlIO_canset_cnt(f)

Implementation can adjust its idea of number of bytes in the buffer. Do not use this – use `PerlIO_fast_gets`.

PerlIO_set_cnt(f,c)

Obscure – set count of bytes in the buffer. Deprecated. Only usable if `PerlIO_canset_cnt()` returns true. Currently used in only `doio.c` to force count less than `-1` to `-1`. Perhaps should be `PerlIO_set_empty` or similar. This call may actually do nothing if "count" is deduced from pointer and a "limit". Do not use this – use `PerlIO_set_ptrcnt()`.

PerlIO_has_base(f)

Returns true if implementation has a buffer, and can return pointer to whole buffer and its size. Used by perl for `-T` / `-B` tests. Other uses would be very obscure...

PerlIO_get_base(f)

Return *start* of buffer. Access only positive offsets in the buffer up to the value returned by `PerlIO_get_bufsiz()`.

PerlIO_get_bufsiz(f)

Return the *total number of bytes* in the buffer, this is neither the number that can be read, nor the amount of memory allocated to the buffer. Rather it is what the operating system and/or implementation happened to `read()` (or whatever) last time IO was requested.

Other Functions**PerlIO_apply_layers(f,mode,layers)**

The new interface to the `USE_PERLIO` implementation. The layers `":crlf"` and `":raw"` are only ones allowed for other implementations and those are silently ignored. (As of perl5.8 `":raw"` is deprecated.) Use `PerlIO_binmode()` below for the portable case.

PerlIO_binmode(f,ptype,imode,layers)

The hook used by perl's `binmode` operator. **ptype** is perl's character for the kind of IO:

```
'<' read
'>' write
'+' read/write
```

imode is `O_BINARY` or `O_TEXT`.

layers is a string of layers to apply, only `":crlf"` makes sense in the non `USE_PERLIO` case. (As of perl5.8 `":raw"` is deprecated in favour of passing `NULL`.)

Portable cases are:

```
PerlIO_binmode(f, ptype, O_BINARY, Nullch);
and
PerlIO_binmode(f, ptype, O_TEXT, ":crlf");
```


On Unix these calls probably have no effect whatsoever. Elsewhere they alter "\n" to CR,LF translation and possibly cause a special text "end of file" indicator to be written or honoured on read. The effect of making the call after doing any IO to the handle depends on the implementation. (It may be ignored, affect any data which is already buffered as well, or only apply to subsequent data.)

PerlIO_debug(fmt,...)

PerlIO_debug is a `printf()`-like function which can be used for debugging. No return value. Its main use is inside PerlIO where using real `printf`, `warn()` etc. would recursively call PerlIO and be a problem.

PerlIO_debug writes to the file named by `$ENV{'PERLIO_DEBUG'}` typical use might be

```
Bourne shells (sh, ksh, bash, zsh, ash, ...):
    PERLIO_DEBUG=/dev/tty ./perl somescript some args
```

```
Csh/Tcsh:
    setenv PERLIO_DEBUG /dev/tty
    ./perl somescript some args
```

```
If you have the "env" utility:
    env PERLIO_DEBUG=/dev/tty ./perl somescript some args
```

```
Win32:
    set PERLIO_DEBUG=CON
    perl somescript some args
```

If `$ENV{'PERLIO_DEBUG'}` is not set `PerlIO_debug()` is a no-op.

NAME

perlbook – Perl book information

DESCRIPTION

The Camel Book, officially known as *Programming Perl, Third Edition*, by Larry Wall et al, is the definitive reference work covering nearly all of Perl. You can order it and other Perl books from O'Reilly & Associates, 1-800-998-9938. Local/overseas is +1 707 829 0515. If you can locate an O'Reilly order form, you can also fax to +1 707 829 0104. If you're web-connected, you can even mosey on over to <http://www.oreilly.com/> for an online order form.

Other Perl books from various publishers and authors can be found listed in [perlfaq2](#).

NAME

perlboot – Beginner’s Object–Oriented Tutorial

DESCRIPTION

If you’re not familiar with objects from other languages, some of the other Perl object documentation may be a little daunting, such as [perlobj](#), a basic reference in using objects, and [perltoot](#), which introduces readers to the peculiarities of Perl’s object system in a tutorial way.

So, let’s take a different approach, presuming no prior object experience. It helps if you know about subroutines ([perlsub](#)), references ([perlref](#) et. seq.), and packages ([perlmod](#)), so become familiar with those first if you haven’t already.

If we could talk to the animals...

Let’s let the animals talk for a moment:

```
sub Cow::speak {
    print "a Cow goes moooo!\n";
}
sub Horse::speak {
    print "a Horse goes neigh!\n";
}
sub Sheep::speak {
    print "a Sheep goes baaaah!\n"
}

Cow::speak;
Horse::speak;
Sheep::speak;
```

This results in:

```
a Cow goes moooo!
a Horse goes neigh!
a Sheep goes baaaah!
```

Nothing spectacular here. Simple subroutines, albeit from separate packages, and called using the full package name. So let’s create an entire pasture:

```
# Cow::speak, Horse::speak, Sheep::speak as before
@pasture = qw(Cow Cow Horse Sheep Sheep);
foreach $animal (@pasture) {
    &{$animal."::speak"};
}
```

This results in:

```
a Cow goes moooo!
a Cow goes moooo!
a Horse goes neigh!
a Sheep goes baaaah!
a Sheep goes baaaah!
```

Wow. That symbolic coderef de-referencing there is pretty nasty. We’re counting on no `strict subs` mode, certainly not recommended for larger programs. And why was that necessary? Because the name of the package seems to be inseparable from the name of the subroutine we want to invoke within that package.

Or is it?

Introducing the method invocation arrow

For now, let's say that `< Class-method` invokes subroutine `method` in package `Class`. (Here, "Class" is used in its "category" meaning, not its "scholastic" meaning.) That's not completely accurate, but we'll do this one step at a time. Now let's use it like so:

```
# Cow::speak, Horse::speak, Sheep::speak as before
Cow->speak;
Horse->speak;
Sheep->speak;
```

And once again, this results in:

```
a Cow goes moooo!
a Horse goes neigh!
a Sheep goes baaaah!
```

That's not fun yet. Same number of characters, all constant, no variables. But yet, the parts are separable now. Watch:

```
$a = "Cow";
$a->speak; # invokes Cow->speak
```

Ahh! Now that the package name has been parted from the subroutine name, we can use a variable package name. And this time, we've got something that works even when `use strict refs` is enabled.

Invoking a barnyard

Let's take that new arrow invocation and put it back in the barnyard example:

```
sub Cow::speak {
    print "a Cow goes moooo!\n";
}
sub Horse::speak {
    print "a Horse goes neigh!\n";
}
sub Sheep::speak {
    print "a Sheep goes baaaah!\n";
}

@pasture = qw(Cow Cow Horse Sheep Sheep);
foreach $animal (@pasture) {
    $animal->speak;
}
```

There! Now we have the animals all talking, and safely at that, without the use of symbolic coderefs.

But look at all that common code. Each of the `speak` routines has a similar structure: a `print` operator and a string that contains common text, except for two of the words. It'd be nice if we could factor out the commonality, in case we decide later to change it all to `says` instead of `goes`.

And we actually have a way of doing that without much fuss, but we have to hear a bit more about what the method invocation arrow is actually doing for us.

The extra parameter of method invocation

The invocation of:

```
Class->method(@args)
```

attempts to invoke subroutine `Class::method` as:

```
Class::method("Class", @args);
```

(If the subroutine can't be found, "inheritance" kicks in, but we'll get to that later.) This means that we get the class name as the first parameter (the only parameter, if no arguments are given). So we can rewrite the Sheep speaking subroutine as:

```
sub Sheep::speak {
    my $class = shift;
    print "a $class goes baaaah!\n";
}
```

And the other two animals come out similarly:

```
sub Cow::speak {
    my $class = shift;
    print "a $class goes moooo!\n";
}
sub Horse::speak {
    my $class = shift;
    print "a $class goes neigh!\n";
}
```

In each case, `$class` will get the value appropriate for that subroutine. But once again, we have a lot of similar structure. Can we factor that out even further? Yes, by calling another method in the same class.

Calling a second method to simplify things

Let's call out from `speak` to a helper method called `sound`. This method provides the constant text for the sound itself.

```
{ package Cow;
  sub sound { "moooo" }
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n"
  }
}
```

Now, when we call `< Cow-speak`, we get a `$class` of `Cow` in `speak`. This in turn selects the `< Cow-sound` method, which returns `moooo`. But how different would this be for the `Horse`?

```
{ package Horse;
  sub sound { "neigh" }
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n"
  }
}
```

Only the name of the package and the specific sound change. So can we somehow share the definition for `speak` between the `Cow` and the `Horse`? Yes, with inheritance!

Inheriting the windpipes

We'll define a common subroutine package called `Animal`, with the definition for `speak`:

```
{ package Animal;
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n"
  }
}
```

Then, for each animal, we say it "inherits" from `Animal`, along with the animal-specific sound:

```
{ package Cow;
  @ISA = qw(Animal);
  sub sound { "moooo" }
}
```

Note the added @ISA array. We'll get to that in a minute.

But what happens when we invoke `< Cow-speak` now?

First, Perl constructs the argument list. In this case, it's just `Cow`. Then Perl looks for `Cow::speak`. But that's not there, so Perl checks for the inheritance array `@Cow::ISA`. It's there, and contains the single name `Animal`.

Perl next checks for `speak` inside `Animal` instead, as in `Animal::speak`. And that's found, so Perl invokes that subroutine with the already frozen argument list.

Inside the `Animal::speak` subroutine, `$class` becomes `Cow` (the first argument). So when we get to the step of invoking `< $class-sound`, it'll be looking for `< Cow-sound`, which gets it on the first try without looking at @ISA. Success!

A few notes about @ISA

This magical @ISA variable (pronounced "is a" not "ice-uh"), has declared that `Cow` "is a" `Animal`. Note that it's an array, not a simple single value, because on rare occasions, it makes sense to have more than one parent class searched for the missing methods.

If `Animal` also had an @ISA, then we'd check there too. The search is recursive, depth-first, left-to-right in each @ISA. Typically, each @ISA has only one element (multiple elements means multiple inheritance and multiple headaches), so we get a nice tree of inheritance.

When we turn on `use strict`, we'll get complaints on @ISA, since it's not a variable containing an explicit package name, nor is it a lexical ("my") variable. We can't make it a lexical variable though (it has to belong to the package to be found by the inheritance mechanism), so there's a couple of straightforward ways to handle that.

The easiest is to just spell the package name out:

```
@Cow::ISA = qw(Animal);
```

Or allow it as an implicitly named package variable:

```
package Cow;
use vars qw(@ISA);
@ISA = qw(Animal);
```

If you're bringing in the class from outside, via an object-oriented module, you change:

```
package Cow;
use Animal;
use vars qw(@ISA);
@ISA = qw(Animal);
```

into just:

```
package Cow;
use base qw(Animal);
```

And that's pretty darn compact.

Overriding the methods

Let's add a mouse, which can barely be heard:

```
# Animal package from before
{ package Mouse;
  @ISA = qw(Animal);
```

```

sub sound { "squeak" }
sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
    print "[but you can barely hear it!]\n";
}
}

Mouse->speak;

```

which results in:

```

a Mouse goes squeak!
[but you can barely hear it!]

```

Here, `Mouse` has its own speaking routine, so `< Mouse-speak` doesn't immediately invoke `< Animal-speak`. This is known as "overriding". In fact, we didn't even need to say that a `Mouse` was an `Animal` at all, since all of the methods needed for `speak` are completely defined with `Mouse`.

But we've now duplicated some of the code from `< Animal-speak`, and this can once again be a maintenance headache. So, can we avoid that? Can we say somehow that a `Mouse` does everything any other `Animal` does, but add in the extra comment? Sure!

First, we can invoke the `Animal::speak` method directly:

```

# Animal package from before
{ package Mouse;
  @ISA = qw(Animal);
  sub sound { "squeak" }
  sub speak {
    my $class = shift;
    Animal::speak($class);
    print "[but you can barely hear it!]\n";
  }
}

```

Note that we have to include the `$class` parameter (almost surely the value of `"Mouse"`) as the first parameter to `Animal::speak`, since we've stopped using the method arrow. Why did we stop? Well, if we invoke `< Animal-speak` there, the first parameter to the method will be `"Animal"` not `"Mouse"`, and when time comes for it to call for the sound, it won't have the right class to come back to this package.

Invoking `Animal::speak` directly is a mess, however. What if `Animal::speak` didn't exist before, and was being inherited from a class mentioned in `@Animal::ISA`? Because we are no longer using the method arrow, we get one and only one chance to hit the right subroutine.

Also note that the `Animal` classname is now hardwired into the subroutine selection. This is a mess if someone maintains the code, changing `@ISA` for `<Mouse` and didn't notice `Animal` there in `speak`. So, this is probably not the right way to go.

Starting the search from a different place

A better solution is to tell Perl to search from a higher place in the inheritance chain:

```

# same Animal as before
{ package Mouse;
  # same @ISA, &sound as before
  sub speak {
    my $class = shift;
    $class->Animal::speak;
    print "[but you can barely hear it!]\n";
  }
}

```

```
}
```

Ahh. This works. Using this syntax, we start with `Animal` to find `speak`, and use all of `Animal`'s inheritance chain if not found immediately. And yet the first parameter will be `$class`, so the found `speak` method will get `Mouse` as its first entry, and eventually work its way back to `Mouse::sound` for the details.

But this isn't the best solution. We still have to keep the `@ISA` and the initial search package coordinated. Worse, if `Mouse` had multiple entries in `@ISA`, we wouldn't necessarily know which one had actually defined `speak`. So, is there an even better way?

The SUPER way of doing things

By changing the `Animal` class to the `SUPER` class in that invocation, we get a search of all of our super classes (classes listed in `@ISA`) automatically:

```
# same Animal as before
{ package Mouse;
  # same @ISA, &sound as before
  sub speak {
    my $class = shift;
    $class->SUPER::speak;
    print "[but you can barely hear it!]\n";
  }
}
```

So, `SUPER::speak` means look in the current package's `@ISA` for `speak`, invoking the first one found.

Where we're at so far...

So far, we've seen the method arrow syntax:

```
Class->method(@args);
```

or the equivalent:

```
$a = "Class";
$a->method(@args);
```

which constructs an argument list of:

```
("Class", @args)
```

and attempts to invoke

```
Class::method("Class", @Args);
```

However, if `Class::method` is not found, then `@Class::ISA` is examined (recursively) to locate a package that does indeed contain `method`, and that subroutine is invoked instead.

Using this simple syntax, we have class methods, (multiple) inheritance, overriding, and extending. Using just what we've seen so far, we've been able to factor out common code, and provide a nice way to reuse implementations with variations. This is at the core of what objects provide, but objects also provide instance data, which we haven't even begun to cover.

A horse is a horse, of course of course — or is it?

Let's start with the code for the `Animal` class and the `Horse` class:

```
{ package Animal;
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n"
  }
}
```



```
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
}
```

This lets us invoke `< Horse->sound` to ripple upward to `Animal::sound`, calling back to `Horse::sound` to get the specific sound, and the output of:

```
a Horse goes neigh!
```

But all of our Horse objects would have to be absolutely identical. If I add a subroutine, all horses automatically share it. That's great for making horses the same, but how do we capture the distinctions about an individual horse? For example, suppose I want to give my first horse a name. There's got to be a way to keep its name separate from the other horses.

We can do that by drawing a new distinction, called an "instance". An "instance" is generally created by a class. In Perl, any reference can be an instance, so let's start with the simplest reference that can hold a horse's name: a scalar reference.

```
my $name = "Mr. Ed";
my $talking = \$name;
```

So now `$talking` is a reference to what will be the instance-specific data (the name). The final step in turning this into a real instance is with a special operator called `bless`:

```
bless $talking, Horse;
```

This operator stores information about the package named `Horse` into the thing pointed at by the reference. At this point, we say `$talking` is an instance of `Horse`. That is, it's a specific horse. The reference is otherwise unchanged, and can still be used with traditional dereferencing operators.

Invoking an instance method

The method arrow can be used on instances, as well as names of packages (classes). So, let's get the sound that `$talking` makes:

```
my $noise = $talking->sound;
```

To invoke `sound`, Perl first notes that `$talking` is a blessed reference (and thus an instance). It then constructs an argument list, in this case from just `($talking)`. (Later we'll see that arguments will take their place following the instance variable, just like with classes.)

Now for the fun part: Perl takes the class in which the instance was blessed, in this case `Horse`, and uses that to locate the subroutine to invoke the method. In this case, `Horse::sound` is found directly (without using inheritance), yielding the final subroutine invocation:

```
Horse::sound($talking)
```

Note that the first parameter here is still the instance, not the name of the class as before. We'll get `neigh` as the return value, and that'll end up as the `$noise` variable above.

If `Horse::sound` had not been found, we'd be wandering up the `@Horse::ISA` list to try to find the method in one of the superclasses, just as for a class method. The only difference between a class method and an instance method is whether the first parameter is an instance (a blessed reference) or a class name (a string).

Accessing the instance data

Because we get the instance as the first parameter, we can now access the instance-specific data. In this case, let's add a way to get at the name:

```
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
  sub name {
    my $self = shift;
```

```
        $$self;
    }
}
```

Now we call for the name:

```
print $talking->name, " says ", $talking->sound, "\n";
```

Inside `Horse::name`, the `@_` array contains just `$talking`, which the `shift` stores into `$self`. (It's traditional to shift the first parameter off into a variable named `$self` for instance methods, so stay with that unless you have strong reasons otherwise.) Then, `$self` gets de-referenced as a scalar ref, yielding `Mr. Ed`, and we're done with that. The result is:

```
Mr. Ed says neigh.
```

How to build a horse

Of course, if we constructed all of our horses by hand, we'd most likely make mistakes from time to time. We're also violating one of the properties of object-oriented programming, in that the "inside guts" of a `Horse` are visible. That's good if you're a veterinarian, but not if you just like to own horses. So, let's let the `Horse` class build a new horse:

```
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
  sub name {
    my $self = shift;
    $$self;
  }
  sub named {
    my $class = shift;
    my $name = shift;
    bless \$name, $class;
  }
}
```

Now with the new named method, we can build a horse:

```
my $talking = Horse->named("Mr. Ed");
```

Notice we're back to a class method, so the two arguments to `Horse::named` are `Horse` and `Mr. Ed`. The `bless` operator not only blesses `$name`, it also returns the reference to `$name`, so that's fine as a return value. And that's how to build a horse.

We've called the constructor `named` here, so that it quickly denotes the constructor's argument as the name for this particular `Horse`. You can use different constructors with different names for different ways of "giving birth" to the object (like maybe recording its pedigree or date of birth). However, you'll find that most people coming to Perl from more limited languages use a single constructor named `new`, with various ways of interpreting the arguments to `new`. Either style is fine, as long as you document your particular way of giving birth to an object. (And you *were* going to do that, right?)

Inheriting the constructor

But was there anything specific to `Horse` in that method? No. Therefore, it's also the same recipe for building anything else that inherited from `Animal`, so let's put it there:

```
{ package Animal;
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n"
  }
  sub name {
```

```

        my $self = shift;
        $$self;
    }
    sub named {
        my $class = shift;
        my $name = shift;
        bless \$name, $class;
    }
}
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
}

```

Ahh, but what happens if we invoke `speak` on an instance?

```

my $talking = Horse->named("Mr. Ed");
$talking->speak;

```

We get a debugging value:

```

a Horse=SCALAR(0xaca42ac) goes neigh!

```

Why? Because the `Animal::speak` routine is expecting a classname as its first parameter, not an instance. When the instance is passed in, we'll end up using a blessed scalar reference as a string, and that shows up as we saw it just now.

Making a method work with either classes or instances

All we need is for a method to detect if it is being called on a class or called on an instance. The most straightforward way is with the `ref` operator. This returns a string (the classname) when used on a blessed reference, and `undef` when used on a string (like a classname). Let's modify the `name` method first to notice the change:

```

sub name {
    my $either = shift;
    ref $either
        ? $$either # it's an instance, return name
        : "an unnamed $either"; # it's a class, return generic
}

```

Here, the `?:` operator comes in handy to select either the dereference or a derived string. Now we can use this with either an instance or a class. Note that I've changed the first parameter holder to `$either` to show that this is intended:

```

my $talking = Horse->named("Mr. Ed");
print Horse->name, "\n"; # prints "an unnamed Horse\n"
print $talking->name, "\n"; # prints "Mr Ed.\n"

```

and now we'll fix `speak` to use this:

```

sub speak {
    my $either = shift;
    print $either->name, " goes ", $either->sound, "\n";
}

```

And since `sound` already worked with either a class or an instance, we're done!

Adding parameters to a method

Let's train our animals to eat:

```

{ package Animal;

```

```

sub named {
    my $class = shift;
    my $name = shift;
    bless \$name, $class;
}
sub name {
    my $either = shift;
    ref $either
        ? $$either # it's an instance, return name
        : "an unnamed $either"; # it's a class, return generic
}
sub speak {
    my $either = shift;
    print $either->name, " goes ", $either->sound, "\n";
}
sub eat {
    my $either = shift;
    my $food = shift;
    print $either->name, " eats $food.\n";
}
}
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
}
{ package Sheep;
  @ISA = qw(Animal);
  sub sound { "baaaah" }
}

```

And now try it out:

```

my $talking = Horse->named("Mr. Ed");
$talking->eat("hay");
Sheep->eat("grass");

```

which prints:

```

Mr. Ed eats hay.
an unnamed Sheep eats grass.

```

An instance method with parameters gets invoked with the instance, and then the list of parameters. So that first invocation is like:

```

Animal::eat($talking, "hay");

```

More interesting instances

What if an instance needs more data? Most interesting instances are made of many items, each of which can in turn be a reference or even another object. The easiest way to store these is often in a hash. The keys of the hash serve as the names of parts of the object (often called "instance variables" or "member variables"), and the corresponding values are, well, the values.

But how do we turn the horse into a hash? Recall that an object was any blessed reference. We can just as easily make it a blessed hash reference as a blessed scalar reference, as long as everything that looks at the reference is changed accordingly.

Let's make a sheep that has a name and a color:

```

my $bad = bless { Name => "Evil", Color => "black" }, Sheep;

```

so `< $bad->{Name}` has Evil, and `< $bad->{Color}` has black. But we want to make `< $bad->name` access the name, and that's now messed up because it's expecting a scalar reference. Not to worry, because that's pretty easy to fix up:

```
## in Animal
sub name {
    my $either = shift;
    ref $either ?
        $either->{Name} :
        "an unnamed $either";
}
```

And of course `named` still builds a scalar sheep, so let's fix that as well:

```
## in Animal
sub named {
    my $class = shift;
    my $name = shift;
    my $self = { Name => $name, Color => $class->default_color };
    bless $self, $class;
}
```

What's this `default_color`? Well, if `named` has only the name, we still need to set a color, so we'll have a class-specific initial color. For a sheep, we might define it as white:

```
## in Sheep
sub default_color { "white" }
```

And then to keep from having to define one for each additional class, we'll define a "backstop" method that serves as the "default default", directly in `Animal`:

```
## in Animal
sub default_color { "brown" }
```

Now, because `name` and `named` were the only methods that referenced the "structure" of the object, the rest of the methods can remain the same, so `speak` still works as before.

A horse of a different color

But having all our horses be brown would be boring. So let's add a method or two to get and set the color.

```
## in Animal
sub color {
    $_[0]->{Color}
}
sub set_color {
    $_[0]->{Color} = $_[1];
}
```

Note the alternate way of accessing the arguments: `$_[0]` is used in-place, rather than with a `shift`. (This saves us a bit of time for something that may be invoked frequently.) And now we can fix that color for Mr. Ed:

```
my $talking = Horse->named("Mr. Ed");
$talking->set_color("black-and-white");
print $talking->name, " is colored ", $talking->color, "\n";
```

which results in:

```
Mr. Ed is colored black-and-white
```

Summary

So, now we have class methods, constructors, instance methods, instance data, and even accessors. But that's still just the beginning of what Perl has to offer. We haven't even begun to talk about accessors that double as getters and setters, destructors, indirect object notation, subclasses that add instance data, per-class data, overloading, "isa" and "can" tests, UNIVERSAL class, and so on. That's for the rest of the Perl documentation to cover. Hopefully, this gets you started, though.

SEE ALSO

For more information, see [perlobj](#) (for all the gritty details about Perl objects, now that you've seen the basics), [perltoot](#) (the tutorial for those who already know objects), [perltooc](#) (dealing with class data), [perlbot](#) (for some more tricks), and books such as Damian Conway's excellent *Object Oriented Perl*.

Some modules which might prove interesting are `Class::Accessor`, `Class::Class`, `Class::Contract`, `Class::Data::Inheritable`, `Class::MethodMaker` and `Tie::SecureHash`

COPYRIGHT

Copyright (c) 1999, 2000 by Randal L. Schwartz and Stonehenge Consulting Services, Inc. Permission is hereby granted to distribute this document intact with the Perl distribution, and in accordance with the licenses of the Perl distribution; derived documents must include this copyright notice intact.

Portions of this text have been derived from Perl Training materials originally appearing in the *Packages, References, Objects, and Modules* course taught by instructors for Stonehenge Consulting Services, Inc. and used with permission.

Portions of this text have been derived from materials originally appearing in *Linux Magazine* and used with permission.

NAME

perlbot – Bag‘o Object Tricks (the BOT)

DESCRIPTION

The following collection of tricks and hints is intended to whet curious appetites about such things as the use of instance variables and the mechanics of object and class relationships. The reader is encouraged to consult relevant textbooks for discussion of Object Oriented definitions and methodology. This is not intended as a tutorial for object-oriented programming or as a comprehensive guide to Perl’s object oriented features, nor should it be construed as a style guide.

The Perl motto still holds: There’s more than one way to do it.

OO SCALING TIPS

- 1 Do not attempt to verify the type of `$self`. That’ll break if the class is inherited, when the type of `$self` is valid but its package isn’t what you expect. See rule 5.
- 2 If an object-oriented (OO) or indirect-object (IO) syntax was used, then the object is probably the correct type and there’s no need to become paranoid about it. Perl isn’t a paranoid language anyway. If people subvert the OO or IO syntax then they probably know what they’re doing and you should let them do it. See rule 1.
- 3 Use the two-argument form of `bless()`. Let a subclass use your constructor. See [INHERITING A CONSTRUCTOR](#).
- 4 The subclass is allowed to know things about its immediate superclass, the superclass is allowed to know nothing about a subclass.
- 5 Don’t be trigger happy with inheritance. A "using", "containing", or "delegation" relationship (some sort of aggregation, at least) is often more appropriate. See [OBJECT RELATIONSHIPS](#), [USING RELATIONSHIP WITH SDBM](#), and ["DELEGATION"](#).
- 6 The object is the namespace. Make package globals accessible via the object. This will remove the guess work about the symbol’s home package. See [CLASS CONTEXT AND THE OBJECT](#).
- 7 IO syntax is certainly less noisy, but it is also prone to ambiguities that can cause difficult-to-find bugs. Allow people to use the sure-thing OO syntax, even if you don’t like it.
- 8 Do not use function-call syntax on a method. You’re going to be bitten someday. Someone might move that method into a superclass and your code will be broken. On top of that you’re feeding the paranoia in rule 2.
- 9 Don’t assume you know the home package of a method. You’re making it difficult for someone to override that method. See [THINKING OF CODE REUSE](#).

INSTANCE VARIABLES

An anonymous array or anonymous hash can be used to hold instance variables. Named parameters are also demonstrated.

```
package Foo;

sub new {
    my $type = shift;
    my %params = @_;
    my $self = {};
    $self->{'High'} = $params{'High'};
    $self->{'Low'}  = $params{'Low'};
    bless $self, $type;
}

package Bar;
```

```

sub new {
    my $type = shift;
    my %params = @_;
    my $self = [];
    $self->[0] = $params{'Left'};
    $self->[1] = $params{'Right'};
    bless $self, $type;
}

package main;

$a = Foo->new( 'High' => 42, 'Low' => 11 );
print "High=$a->{'High'}\n";
print "Low=$a->{'Low'}\n";

$b = Bar->new( 'Left' => 78, 'Right' => 40 );
print "Left=$b->[0]\n";
print "Right=$b->[1]\n";

```

SCALAR INSTANCE VARIABLES

An anonymous scalar can be used when only one instance variable is needed.

```

package Foo;

sub new {
    my $type = shift;
    my $self;
    $self = shift;
    bless \$self, $type;
}

package main;

$a = Foo->new( 42 );
print "a=$$a\n";

```

INSTANCE VARIABLE INHERITANCE

This example demonstrates how one might inherit instance variables from a superclass for inclusion in the new class. This requires calling the superclass's constructor and adding one's own instance variables to the new object.

```

package Bar;

sub new {
    my $type = shift;
    my $self = {};
    $self->{'buz'} = 42;
    bless $self, $type;
}

package Foo;
@ISA = qw( Bar );

sub new {
    my $type = shift;
    my $self = Bar->new;
    $self->{'biz'} = 11;
    bless $self, $type;
}

package main;

```



```
$a = Foo->new;
print "buz = ", $a->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

OBJECT RELATIONSHIPS

The following demonstrates how one might implement "containing" and "using" relationships between objects.

```
package Bar;

sub new {
    my $type = shift;
    my $self = {};
    $self->{'buz'} = 42;
    bless $self, $type;
}

package Foo;

sub new {
    my $type = shift;
    my $self = {};
    $self->{'Bar'} = Bar->new;
    $self->{'biz'} = 11;
    bless $self, $type;
}

package main;

$a = Foo->new;
print "buz = ", $a->{'Bar'}->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

OVERRIDING SUPERCLASS METHODS

The following example demonstrates how to override a superclass method and then call the overridden method. The **SUPER** pseudo-class allows the programmer to call an overridden superclass method without actually knowing where that method is defined.

```
package Buz;
sub goo { print "here's the goo\n" }

package Bar; @ISA = qw( Buz );
sub google { print "google here\n" }

package Baz;
sub mumble { print "mumbling\n" }

package Foo;
@ISA = qw( Bar Baz );

sub new {
    my $type = shift;
    bless [], $type;
}

sub grr { print "grumble\n" }
sub goo {
    my $self = shift;
    $self->SUPER::goo();
}

sub mumble {
    my $self = shift;
```

```

        $self->SUPER::mumble();
    }
    sub google {
        my $self = shift;
        $self->SUPER::google();
    }

    package main;

    $foo = Foo->new;
    $foo->mumble;
    $foo->grr;
    $foo->goo;
    $foo->google;

```

USING RELATIONSHIP WITH SDBM

This example demonstrates an interface for the SDBM class. This creates a "using" relationship between the SDBM class and the new class Mydbm.

```

package Mydbm;

require SDBM_File;
require Tie::Hash;
@ISA = qw( Tie::Hash );

sub TIEHASH {
    my $type = shift;
    my $ref = SDBM_File->new(@_);
    bless {'dbm' => $ref}, $type;
}

sub FETCH {
    my $self = shift;
    my $ref = $self->{'dbm'};
    $ref->FETCH(@_);
}

sub STORE {
    my $self = shift;
    if (defined $_[0]){
        my $ref = $self->{'dbm'};
        $ref->STORE(@_);
    } else {
        die "Cannot STORE an undefined key in Mydbm\n";
    }
}

package main;

use Fcntl qw( O_RDWR O_CREAT );

tie %foo, "Mydbm", "Sdbm", O_RDWR|O_CREAT, 0640;
$foo{'bar'} = 123;
print "foo-bar = $foo{'bar'}\n";

tie %bar, "Mydbm", "Sdbm2", O_RDWR|O_CREAT, 0640;
$bar{'Cathy'} = 456;
print "bar-Cathy = $bar{'Cathy'}\n";

```

THINKING OF CODE REUSE

One strength of Object-Oriented languages is the ease with which old code can use new code. The following examples will demonstrate first how one can hinder code reuse and then how one can promote code reuse.

This first example illustrates a class which uses a fully-qualified method call to access the "private" method `BAZ()`. The second example will show that it is impossible to override the `BAZ()` method.

```
package FOO;

sub new {
    my $type = shift;
    bless {}, $type;
}

sub bar {
    my $self = shift;
    $self->FOO::private::BAZ;
}

package FOO::private;

sub BAZ {
    print "in BAZ\n";
}

package main;

$a = FOO->new;
$a->bar;
```

Now we try to override the `BAZ()` method. We would like `FOO::bar()` to call `GOOP::BAZ()`, but this cannot happen because `FOO::bar()` explicitly calls `FOO::private::BAZ()`.

```
package FOO;

sub new {
    my $type = shift;
    bless {}, $type;
}

sub bar {
    my $self = shift;
    $self->FOO::private::BAZ;
}

package FOO::private;

sub BAZ {
    print "in BAZ\n";
}

package GOOP;
@ISA = qw( FOO );
sub new {
    my $type = shift;
    bless {}, $type;
}

sub BAZ {
    print "in GOOP::BAZ\n";
}
```

```
package main;

$a = GOOP->new;
$a->bar;
```

To create reusable code we must modify class FOO, flattening class FOO::private. The next example shows a reusable class FOO which allows the method GOOP::BAZ() to be used in place of FOO::BAZ().

```
package FOO;

sub new {
    my $type = shift;
    bless {}, $type;
}

sub bar {
    my $self = shift;
    $self->BAZ;
}

sub BAZ {
    print "in BAZ\n";
}

package GOOP;
@ISA = qw( FOO );

sub new {
    my $type = shift;
    bless {}, $type;
}

sub BAZ {
    print "in GOOP::BAZ\n";
}

package main;

$a = GOOP->new;
$a->bar;
```

CLASS CONTEXT AND THE OBJECT

Use the object to solve package and class context problems. Everything a method needs should be available via the object or should be passed as a parameter to the method.

A class will sometimes have static or global data to be used by the methods. A subclass may want to override that data and replace it with new data. When this happens the superclass may not know how to find the new copy of the data.

This problem can be solved by using the object to define the context of the method. Let the method look in the object for a reference to the data. The alternative is to force the method to go hunting for the data ("Is it in my class, or in a subclass? Which subclass?"), and this can be inconvenient and will lead to hackery. It is better just to let the object tell the method where that data is located.

```
package Bar;

%fizzle = ( 'Password' => 'XYZZY' );

sub new {
    my $type = shift;
    my $self = {};
    $self->{'fizzle'} = \%fizzle;
    bless $self, $type;
}
```

```

sub enter {
    my $self = shift;

    # Don't try to guess if we should use %Bar::fizzle
    # or %Foo::fizzle. The object already knows which
    # we should use, so just ask it.
    #
    my $fizzle = $self->{'fizzle'};

    print "The word is ", $fizzle->{'Password'}, "\n";
}

package Foo;
@ISA = qw( Bar );

%fizzle = ( 'Password' => 'Rumple' );

sub new {
    my $type = shift;
    my $self = Bar->new;
    $self->{'fizzle'} = \%fizzle;
    bless $self, $type;
}

package main;

$a = Bar->new;
$b = Foo->new;
$a->enter;
$b->enter;

```

INHERITING A CONSTRUCTOR

An inheritable constructor should use the second form of `bless()` which allows blessing directly into a specified class. Notice in this example that the object will be a `BAR` not a `FOO`, even though the constructor is in class `FOO`.

```

package FOO;

sub new {
    my $type = shift;
    my $self = {};
    bless $self, $type;
}

sub baz {
    print "in FOO::baz()\n";
}

package BAR;
@ISA = qw( FOO );

sub baz {
    print "in BAR::baz()\n";
}

package main;

$a = BAR->new;
$a->baz;

```

DELEGATION

Some classes, such as `SDBM_File`, cannot be effectively subclassed because they create foreign objects. Such a class can be extended with some sort of aggregation technique such as the "using" relationship mentioned earlier or by delegation.

The following example demonstrates delegation using an `AUTOLOAD()` function to perform message-forwarding. This will allow the `Mydbm` object to behave exactly like an `SDBM_File` object. The `Mydbm` class could now extend the behavior by adding custom `FETCH()` and `STORE()` methods, if this is desired.

```
package Mydbm;

require SDBM_File;
require Tie::Hash;
@ISA = qw(Tie::Hash);

sub TIEHASH {
    my $type = shift;
    my $ref = SDBM_File->new(@_);
    bless {'delegate' => $ref};
}

sub AUTOLOAD {
    my $self = shift;

    # The Perl interpreter places the name of the
    # message in a variable called $AUTOLOAD.

    # DESTROY messages should never be propagated.
    return if $AUTOLOAD =~ /::DESTROY$/;

    # Remove the package name.
    $AUTOLOAD =~ s/^Mydbm:://;

    # Pass the message to the delegate.
    $self->{'delegate'}->$AUTOLOAD(@_);
}

package main;
use Fcntl qw( O_RDWR O_CREAT );

tie %foo, "Mydbm", "adbm", O_RDWR|O_CREAT, 0640;
$foo{'bar'} = 123;
print "foo-bar = $foo{'bar'}\n";
```

NAME

perlcall – Perl calling conventions from C

DESCRIPTION

The purpose of this document is to show you how to call Perl subroutines directly from C, i.e., how to write *callbacks*.

Apart from discussing the C interface provided by Perl for writing callbacks the document uses a series of examples to show how the interface actually works in practice. In addition some techniques for coding callbacks are covered.

Examples where callbacks are necessary include

- An Error Handler

You have created an XSUB interface to an application's C API.

A fairly common feature in applications is to allow you to define a C function that will be called whenever something nasty occurs. What we would like is to be able to specify a Perl subroutine that will be called instead.

- An Event Driven Program

The classic example of where callbacks are used is when writing an event driven program like for an X windows application. In this case you register functions to be called whenever specific events occur, e.g., a mouse button is pressed, the cursor moves into a window or a menu item is selected.

Although the techniques described here are applicable when embedding Perl in a C program, this is not the primary goal of this document. There are other details that must be considered and are specific to embedding Perl. For details on embedding Perl in C refer to [perlembed](#).

Before you launch yourself head first into the rest of this document, it would be a good idea to have read the following two documents – [perlxs](#) and [perlguts](#).

THE CALL_ FUNCTIONS

Although this stuff is easier to explain using examples, you first need be aware of a few important definitions.

Perl has a number of C functions that allow you to call Perl subroutines. They are

```
I32 call_sv(SV* sv, I32 flags) ;
I32 call_pv(char *subname, I32 flags) ;
I32 call_method(char *methname, I32 flags) ;
I32 call_argv(char *subname, I32 flags, register char **argv) ;
```

The key function is *call_sv*. All the other functions are fairly simple wrappers which make it easier to call Perl subroutines in special cases. At the end of the day they will all call *call_sv* to invoke the Perl subroutine.

All the *call_** functions have a *flags* parameter which is used to pass a bit mask of options to Perl. This bit mask operates identically for each of the functions. The settings available in the bit mask are discussed in [FLAG VALUES](#).

Each of the functions will now be discussed in turn.

call_sv

call_sv takes two parameters, the first, *sv*, is an SV*. This allows you to specify the Perl subroutine to be called either as a C string (which has first been converted to an SV) or a reference to a subroutine. The section, *Using call_sv*, shows how you can make use of *call_sv*.

call_pv

The function, *call_pv*, is similar to *call_sv* except it expects its first parameter to be a C char* which identifies the Perl subroutine you want to call, e.g., `call_pv("fred", 0)`. If the subroutine you

want to call is in another package, just include the package name in the string, e.g., "pkg::fred".

call_method

The function *call_method* is used to call a method from a Perl class. The parameter *methname* corresponds to the name of the method to be called. Note that the class that the method belongs to is passed on the Perl stack rather than in the parameter list. This class can be either the name of the class (for a static method) or a reference to an object (for a virtual method). See [perlobj](#) for more information on static and virtual methods and [Using call_method](#) for an example of using *call_method*.

call_argv

call_argv calls the Perl subroutine specified by the C string stored in the *subname* parameter. It also takes the usual *flags* parameter. The final parameter, *argv*, consists of a NULL terminated list of C strings to be passed as parameters to the Perl subroutine. See [Using call_argv](#).

All the functions return an integer. This is a count of the number of items returned by the Perl subroutine. The actual items returned by the subroutine are stored on the Perl stack.

As a general rule you should *always* check the return value from these functions. Even if you are expecting only a particular number of values to be returned from the Perl subroutine, there is nothing to stop someone from doing something unexpected—don't say you haven't been warned.

FLAG VALUES

The *flags* parameter in all the *call_** functions is a bit mask which can consist of any combination of the symbols defined below, OR'ed together.

G_VOID

Calls the Perl subroutine in a void context.

This flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a void context (if it executes *wantarray* the result will be the undefined value).
2. It ensures that nothing is actually returned from the subroutine.

The value returned by the *call_** function indicates how many items have been returned by the Perl subroutine – in this case it will be 0.

G_SCALAR

Calls the Perl subroutine in a scalar context. This is the default context flag setting for all the *call_** functions.

This flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a scalar context (if it executes *wantarray* the result will be false).
2. It ensures that only a scalar is actually returned from the subroutine. The subroutine can, of course, ignore the *wantarray* and return a list anyway. If so, then only the last element of the list will be returned.

The value returned by the *call_** function indicates how many items have been returned by the Perl subroutine – in this case it will be either 0 or 1.

If 0, then you have specified the *G_DISCARD* flag.

If 1, then the item actually returned by the Perl subroutine will be stored on the Perl stack – the section *Returning a Scalar* shows how to access this value on the stack. Remember that regardless of how many items the Perl subroutine returns, only the last one will be accessible from the stack – think of the case where only one value is returned as being a list with only one element. Any other items that were returned will not exist by the time control returns from the *call_** function. The section *Returning a list in a scalar context*

shows an example of this behavior.

G_ARRAY

Calls the Perl subroutine in a list context.

As with G_SCALAR, this flag has 2 effects:

1. It indicates to the subroutine being called that it is executing in a list context (if it executes *wantarray* the result will be true).
2. It ensures that all items returned from the subroutine will be accessible when control returns from the *call_** function.

The value returned by the *call_** function indicates how many items have been returned by the Perl subroutine.

If 0, then you have specified the G_DISCARD flag.

If not 0, then it will be a count of the number of items returned by the subroutine. These items will be stored on the Perl stack. The section *Returning a list of values* gives an example of using the G_ARRAY flag and the mechanics of accessing the returned items from the Perl stack.

G_DISCARD

By default, the *call_** functions place the items returned from by the Perl subroutine on the stack. If you are not interested in these items, then setting this flag will make Perl get rid of them automatically for you. Note that it is still possible to indicate a context to the Perl subroutine by using either G_SCALAR or G_ARRAY.

If you do not set this flag then it is *very* important that you make sure that any temporaries (i.e., parameters passed to the Perl subroutine and values returned from the subroutine) are disposed of yourself. The section *Returning a Scalar* gives details of how to dispose of these temporaries explicitly and the section *Using Perl to dispose of temporaries* discusses the specific circumstances where you can ignore the problem and let Perl deal with it for you.

G_NOARGS

Whenever a Perl subroutine is called using one of the *call_** functions, it is assumed by default that parameters are to be passed to the subroutine. If you are not passing any parameters to the Perl subroutine, you can save a bit of time by setting this flag. It has the effect of not creating the `@_` array for the Perl subroutine.

Although the functionality provided by this flag may seem straightforward, it should be used only if there is a good reason to do so. The reason for being cautious is that even if you have specified the G_NOARGS flag, it is still possible for the Perl subroutine that has been called to think that you have passed it parameters.

In fact, what can happen is that the Perl subroutine you have called can access the `@_` array from a previous Perl subroutine. This will occur when the code that is executing the *call_** function has itself been called from another Perl subroutine. The code below illustrates this

```
sub fred
{ print "@_\n" }

sub joe
{ &fred }

&joe(1,2,3) ;
```

This will print

```
1 2 3
```

What has happened is that `fred` accesses the `@_` array which belongs to `joe`.

G_EVAL

It is possible for the Perl subroutine you are calling to terminate abnormally, e.g., by calling *die* explicitly or by not actually existing. By default, when either of these events occurs, the process will terminate immediately. If you want to trap this type of event, specify the `G_EVAL` flag. It will put an *eval { }* around the subroutine call.

Whenever control returns from the *call_** function you need to check the `$@` variable as you would in a normal Perl script.

The value returned from the *call_** function is dependent on what other flags have been specified and whether an error has occurred. Here are all the different cases that can occur:

- If the *call_** function returns normally, then the value returned is as specified in the previous sections.
- If `G_DISCARD` is specified, the return value will always be 0.
- If `G_ARRAY` is specified *and* an error has occurred, the return value will always be 0.
- If `G_SCALAR` is specified *and* an error has occurred, the return value will be 1 and the value on the top of the stack will be *undef*. This means that if you have already detected the error by checking `$@` and you want the program to continue, you must remember to pop the *undef* from the stack.

See *Using G_EVAL* for details on using `G_EVAL`.

G_KEEPPERR

You may have noticed that using the `G_EVAL` flag described above will **always** clear the `$@` variable and set it to a string describing the error iff there was an error in the called code. This unqualified resetting of `$@` can be problematic in the reliable identification of errors using the *eval { }* mechanism, because the possibility exists that perl will call other code (end of block processing code, for example) between the time the error causes `$@` to be set within *eval { }*, and the subsequent statement which checks for the value of `$@` gets executed in the user's script.

This scenario will mostly be applicable to code that is meant to be called from within destructors, asynchronous callbacks, signal handlers, `__DIE__` or `__WARN__` hooks, and *tie* functions. In such situations, you will not want to clear `$@` at all, but simply to append any new errors to any existing value of `$@`.

The `G_KEEPPERR` flag is meant to be used in conjunction with `G_EVAL` in *call_** functions that are used to implement such code. This flag has no effect when `G_EVAL` is not used.

When `G_KEEPPERR` is used, any errors in the called code will be prefixed with the string `"\t(in cleanup)"`, and appended to the current value of `$@`.

The `G_KEEPPERR` flag was introduced in Perl version 5.002.

See *Using G_KEEPPERR* for an example of a situation that warrants the use of this flag.

Determining the Context

As mentioned above, you can determine the context of the currently executing subroutine in Perl with *wantarray*. The equivalent test can be made in C by using the `GIMME_V` macro, which returns `G_ARRAY` if you have been called in a list context, `G_SCALAR` if in a scalar context, or `G_VOID` if in a void context (i.e. the return value will not be used). An older version of this macro is called `GIMME`; in a void context it returns `G_SCALAR` instead of `G_VOID`. An example of using the `GIMME_V` macro is shown in section *Using GIMME_V*.

KNOWN PROBLEMS

This section outlines all known problems that exist in the *call_** functions.

1. If you are intending to make use of both the `G_EVAL` and `G_SCALAR` flags in your code, use a version of Perl greater than 5.000. There is a bug in version 5.000 of Perl which means that the combination of these two flags will not work as described in the section *FLAG VALUES*.

Specifically, if the two flags are used when calling a subroutine and that subroutine does not call *die*, the value returned by *call_** will be wrong.

2. In Perl 5.000 and 5.001 there is a problem with using *call_** if the Perl sub you are calling attempts to trap a *die*.

The symptom of this problem is that the called Perl sub will continue to completion, but whenever it attempts to pass control back to the XSUB, the program will immediately terminate.

For example, say you want to call this Perl sub

```
sub fred
{
    eval { die "Fatal Error" ; }
    print "Trapped error: $@\n"
    if $@ ;
}
```

via this XSUB

```
void
Call_fred()
CODE:
    PUSHMARK(SP) ;
    call_pv("fred", G_DISCARD|G_NOARGS) ;
    fprintf(stderr, "back in Call_fred\n") ;
```

When *Call_fred* is executed it will print

```
Trapped error: Fatal Error
```

As control never returns to *Call_fred*, the "back in *Call_fred*" string will not get printed.

To work around this problem, you can either upgrade to Perl 5.002 or higher, or use the `G_EVAL` flag with *call_** as shown below

```
void
Call_fred()
CODE:
    PUSHMARK(SP) ;
    call_pv("fred", G_EVAL|G_DISCARD|G_NOARGS) ;
    fprintf(stderr, "back in Call_fred\n") ;
```

EXAMPLES

Enough of the definition talk, let's have a few examples.

Perl provides many macros to assist in accessing the Perl stack. Wherever possible, these macros should always be used when interfacing to Perl internals. We hope this should make the code less vulnerable to any changes made to Perl in the future.

Another point worth noting is that in the first series of examples I have made use of only the *call_pv* function. This has been done to keep the code simpler and ease you into the topic. Wherever possible, if the choice is between using *call_pv* and *call_sv*, you should always try to use *call_sv*. See *Using call_sv* for details.

No Parameters, Nothing returned

This first trivial example will call a Perl subroutine, *PrintUID*, to print out the UID of the process.

```
sub PrintUID
{
    print "UID is $<\n" ;
}
```

and here is a C function to call it

```
static void
call_PrintUID()
{
    dSP ;

    PUSHMARK(SP) ;
    call_pv("PrintUID", G_DISCARD|G_NOARGS) ;
}
```

Simple, eh.

A few points to note about this example.

1. Ignore `dSP` and `PUSHMARK(SP)` for now. They will be discussed in the next example.
2. We aren't passing any parameters to *PrintUID* so `G_NOARGS` can be specified.
3. We aren't interested in anything returned from *PrintUID*, so `G_DISCARD` is specified. Even if *PrintUID* was changed to return some value(s), having specified `G_DISCARD` will mean that they will be wiped by the time control returns from *call_pv*.
4. As *call_pv* is being used, the Perl subroutine is specified as a C string. In this case the subroutine name has been 'hard-wired' into the code.
5. Because we specified `G_DISCARD`, it is not necessary to check the value returned from *call_pv*. It will always be 0.

Passing Parameters

Now let's make a slightly more complex example. This time we want to call a Perl subroutine, *LeftString*, which will take 2 parameters—a string (`$s`) and an integer (`$n`) . The subroutine will simply print the first `$n` characters of the string.

So the Perl subroutine would look like this

```
sub LeftString
{
    my($s, $n) = @_ ;
    print substr($s, 0, $n), "\n" ;
}
```

The C function required to call *LeftString* would look like this.

```
static void
call_LeftString(a, b)
char * a ;
int b ;
{
    dSP ;

    ENTER ;
    SAVETMPS ;
```

```

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSVpv(a, 0)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK ;

    call_pv("LeftString", G_DISCARD);

    FREETMPS ;
    LEAVE ;
}

```

Here are a few notes on the C function *call_LeftString*.

1. Parameters are passed to the Perl subroutine using the Perl stack. This is the purpose of the code beginning with the line `dSP` and ending with the line `PUTBACK`. The `dSP` declares a local copy of the stack pointer. This local copy should **always** be accessed as `SP`.
2. If you are going to put something onto the Perl stack, you need to know where to put it. This is the purpose of the macro `dSP`—it declares and initializes a *local* copy of the Perl stack pointer.

All the other macros which will be used in this example require you to have used this macro.

The exception to this rule is if you are calling a Perl subroutine directly from an `XSUB` function. In this case it is not necessary to use the `dSP` macro explicitly—it will be declared for you automatically.

3. Any parameters to be pushed onto the stack should be bracketed by the `PUSHMARK` and `PUTBACK` macros. The purpose of these two macros, in this context, is to count the number of parameters you are pushing automatically. Then whenever Perl is creating the `@_` array for the subroutine, it knows how big to make it.

The `PUSHMARK` macro tells Perl to make a mental note of the current stack pointer. Even if you aren't passing any parameters (like the example shown in the section *No Parameters, Nothing returned*) you must still call the `PUSHMARK` macro before you can call any of the *call_** functions—Perl still needs to know that there are no parameters.

The `PUTBACK` macro sets the global copy of the stack pointer to be the same as our local copy. If we didn't do this *call_pv* wouldn't know where the two parameters we pushed were—remember that up to now all the stack pointer manipulation we have done is with our local copy, *not* the global copy.

4. Next, we come to `XPUSHs`. This is where the parameters actually get pushed onto the stack. In this case we are pushing a string and an integer.

See *XSUBs and the Argument Stack in perl guts* for details on how the `XPUSH` macros work.

5. Because we created temporary values (by means of `sv_2mortal()` calls) we will have to tidy up the Perl stack and dispose of mortal SVs.

This is the purpose of

```

    ENTER ;
    SAVETMPS ;

```

at the start of the function, and

```

    FREETMPS ;
    LEAVE ;

```

at the end. The `ENTER/SAVETMPS` pair creates a boundary for any temporaries we create. This means that the temporaries we get rid of will be limited to those which were created after these calls.

The `FREETMPS/LEAVE` pair will get rid of any values returned by the Perl subroutine (see next example), plus it will also dump the mortal SVs we have created. Having `ENTER/SAVETMPS` at the beginning of the code makes sure that no other mortals are destroyed.

Think of these macros as working a bit like using { and } in Perl to limit the scope of local variables.

See the section *Using Perl to dispose of temporaries* for details of an alternative to using these macros.

6. Finally, *LeftString* can now be called via the *call_pv* function. The only flag specified this time is *G_DISCARD*. Because we are passing 2 parameters to the Perl subroutine this time, we have not specified *G_NOARGS*.

Returning a Scalar

Now for an example of dealing with the items returned from a Perl subroutine.

Here is a Perl subroutine, *Adder*, that takes 2 integer parameters and simply returns their sum.

```
sub Adder
{
    my($a, $b) = @_ ;
    $a + $b ;
}
```

Because we are now concerned with the return value from *Adder*, the C function required to call it is now a bit more complex.

```
static void
call_Adder(a, b)
int a ;
int b ;
{
    dSP ;
    int count ;

    ENTER ;
    SAVETMPS;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(a))) ;
    XPUSHs(sv_2mortal(newSViv(b))) ;
    PUTBACK ;

    count = call_pv("Adder", G_SCALAR);

    SPAGAIN ;

    if (count != 1)
        croak("Big trouble\n") ;

    printf ("The sum of %d and %d is %d\n", a, b, POPi) ;

    PUTBACK ;
    FREETMPS ;
    LEAVE ;
}
```

Points to note this time are

1. The only flag specified this time was *G_SCALAR*. That means the *@_* array will be created and that the value returned by *Adder* will still exist after the call to *call_pv*.
2. The purpose of the macro *SPAGAIN* is to refresh the local copy of the stack pointer. This is necessary because it is possible that the memory allocated to the Perl stack has been reallocated whilst in the *call_pv* call.

If you are making use of the Perl stack pointer in your code you must always refresh the local copy using SPAGAIN whenever you make use of the *call_** functions or any other Perl internal function.

3. Although only a single value was expected to be returned from *Adder*, it is still good practice to check the return code from *call_pv* anyway.

Expecting a single value is not quite the same as knowing that there will be one. If someone modified *Adder* to return a list and we didn't check for that possibility and take appropriate action the Perl stack would end up in an inconsistent state. That is something you *really* don't want to happen ever.

4. The *POPi* macro is used here to pop the return value from the stack. In this case we wanted an integer, so *POPi* was used.

Here is the complete list of POP macros available, along with the types they return.

POPs	SV
POPP	pointer
POPn	double
POPi	integer
POP1	long

5. The final *PUTBACK* is used to leave the Perl stack in a consistent state before exiting the function. This is necessary because when we popped the return value from the stack with *POPi* it updated only our local copy of the stack pointer. Remember, *PUTBACK* sets the global stack pointer to be the same as our local copy.

Returning a list of values

Now, let's extend the previous example to return both the sum of the parameters and the difference.

Here is the Perl subroutine

```
sub AddSubtract
{
    my($a, $b) = @_ ;
    ($a+$b, $a-$b) ;
}
```

and this is the C function

```
static void
call_AddSubtract(a, b)
int a ;
int b ;
{
    dSP ;
    int count ;

    ENTER ;
    SAVETMPS;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(a))) ;
    XPUSHs(sv_2mortal(newSViv(b))) ;
    PUTBACK ;

    count = call_pv("AddSubtract", G_ARRAY);

    SPAGAIN ;

    if (count != 2)
        croak("Big trouble\n") ;
```

```

        printf ("%d - %d = %d\n", a, b, POPi) ;
        printf ("%d + %d = %d\n", a, b, POPi) ;

        PUTBACK ;
        FREETMPS ;
        LEAVE ;
    }

```

If `call_AddSubtract` is called like this

```
call_AddSubtract(7, 4) ;
```

then here is the output

```

7 - 4 = 3
7 + 4 = 11

```

Notes

1. We wanted list context, so `G_ARRAY` was used.
2. Not surprisingly `POPi` is used twice this time because we were retrieving 2 values from the stack. The important thing to note is that when using the `POP*` macros they come off the stack in *reverse* order.

Returning a list in a scalar context

Say the Perl subroutine in the previous section was called in a scalar context, like this

```

static void
call_AddSubScalar(a, b)
int a ;
int b ;
{
    dSP ;
    int count ;
    int i ;

    ENTER ;
    SAVETMPS ;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(a))) ;
    XPUSHs(sv_2mortal(newSViv(b))) ;
    PUTBACK ;

    count = call_pv("AddSubtract", G_SCALAR) ;

    SPAGAIN ;

    printf ("Items Returned = %d\n", count) ;
    for (i = 1 ; i <= count ; ++i)
        printf ("Value %d = %d\n", i, POPi) ;

    PUTBACK ;
    FREETMPS ;
    LEAVE ;
}

```

The other modification made is that `call_AddSubScalar` will print the number of items returned from the Perl subroutine and their value (for simplicity it assumes that they are integer). So if `call_AddSubScalar` is called


```
call_AddSubScalar(7, 4) ;
```

then the output will be

```
Items Returned = 1
Value 1 = 3
```

In this case the main point to note is that only the last item in the list is returned from the subroutine, *AddSubtract* actually made it back to *call_AddSubScalar*.

Returning Data from Perl via the parameter list

It is also possible to return values directly via the parameter list – whether it is actually desirable to do it is another matter entirely.

The Perl subroutine, *Inc*, below takes 2 parameters and increments each directly.

```
sub Inc
{
    ++ $_[0] ;
    ++ $_[1] ;
}
```

and here is a C function to call it.

```
static void
call_Inc(a, b)
int a ;
int b ;
{
    dSP ;
    int count ;
    SV * sva ;
    SV * svb ;

    ENTER ;
    SAVETMPS ;

    sva = sv_2mortal(newSViv(a)) ;
    svb = sv_2mortal(newSViv(b)) ;

    PUSHMARK(SP) ;
    XPUSHs(sva) ;
    XPUSHs(svb) ;
    PUTBACK ;

    count = call_pv("Inc", G_DISCARD) ;

    if (count != 0)
        croak ("call_Inc: expected 0 values from 'Inc', got %d\n",
              count) ;

    printf ("%d + 1 = %d\n", a, SvIV(sva)) ;
    printf ("%d + 1 = %d\n", b, SvIV(svb)) ;

    FREETMPS ;
    LEAVE ;
}
```

To be able to access the two parameters that were pushed onto the stack after they return from *call_pv* it is necessary to make a note of their addresses—thus the two variables *sva* and *svb*.

The reason this is necessary is that the area of the Perl stack which held them will very likely have been overwritten by something else by the time control returns from *call_pv*.

Using G_EVAL

Now an example using G_EVAL. Below is a Perl subroutine which computes the difference of its 2 parameters. If this would result in a negative result, the subroutine calls *die*.

```
sub Subtract
{
    my ($a, $b) = @_ ;
    die "death can be fatal\n" if $a < $b ;
    $a - $b ;
}
```

and some C to call it

```
static void
call_Subtract(a, b)
int a ;
int b ;
{
    dSP ;
    int count ;

    ENTER ;
    SAVETMPS;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(a))) ;
    XPUSHs(sv_2mortal(newSViv(b))) ;
    PUTBACK ;

    count = call_pv("Subtract", G_EVAL|G_SCALAR);

    SPAGAIN ;

    /* Check the eval first */
    if (SvTRUE(ERRSV))
    {
        STRLEN n_a;
        printf ("Uh oh - %s\n", SvPV(ERRSV, n_a)) ;
        POPs ;
    }
    else
    {
        if (count != 1)
            croak("call_Subtract: wanted 1 value from 'Subtract', got %d\n",
                count) ;

        printf ("%d - %d = %d\n", a, b, POPi) ;
    }

    PUTBACK ;
    FREETMPS ;
    LEAVE ;
}
```

If *call_Subtract* is called thus

```
call_Subtract(4, 5)
```

the following will be printed

Uh oh - death can be fatal

Notes

1. We want to be able to catch the *die* so we have used the `G_EVAL` flag. Not specifying this flag would mean that the program would terminate immediately at the *die* statement in the subroutine *Subtract*.
2. The code

```
if (SvTRUE(ERRSV))
{
    STRLEN n_a;
    printf ("Uh oh - %s\n", SvPV(ERRSV, n_a)) ;
    POPs ;
}
```

is the direct equivalent of this bit of Perl

```
print "Uh oh - $@\n" if $@ ;
```

`PL_errgv` is a perl global of type `GV *` that points to the symbol table entry containing the error. `ERRSV` therefore refers to the C equivalent of `$@`.

3. Note that the stack is popped using `POPs` in the block where `SvTRUE(ERRSV)` is true. This is necessary because whenever a *call_** function invoked with `G_EVAL|G_SCALAR` returns an error, the top of the stack holds the value *undef*. Because we want the program to continue after detecting this error, it is essential that the stack is tidied up by removing the *undef*.

Using `G_KEEPPERR`

Consider this rather facetious example, where we have used an XS version of the `call_Subtract` example above inside a destructor:

```
package Foo;
sub new { bless {}, $_[0] }
sub Subtract {
    my($a,$b) = @_;
    die "death can be fatal" if $a < $b ;
    $a - $b;
}
sub DESTROY { call_Subtract(5, 4); }
sub foo { die "foo dies"; }

package main;
eval { Foo->new->foo };
print "Saw: $@" if $@;                # should be, but isn't
```

This example will fail to recognize that an error occurred inside the `eval {}`. Here's why: the `call_Subtract` code got executed while perl was cleaning up temporaries when exiting the `eval` block, and because `call_Subtract` is implemented with *call_pv* using the `G_EVAL` flag, it promptly reset `$@`. This results in the failure of the outermost test for `$@`, and thereby the failure of the error trap.

Appending the `G_KEEPPERR` flag, so that the *call_pv* call in `call_Subtract` reads:

```
count = call_pv("Subtract", G_EVAL|G_SCALAR|G_KEEPPERR);
```

will preserve the error and restore reliable error handling.

Using `call_sv`

In all the previous examples I have 'hard-wired' the name of the Perl subroutine to be called from C. Most of the time though, it is more convenient to be able to specify the name of the Perl subroutine from within the Perl script.

Consider the Perl code below

```
sub fred
{
    print "Hello there\n" ;
}

CallSubPV("fred") ;
```

Here is a snippet of XSUB which defines *CallSubPV*.

```
void
CallSubPV(name)
    char *   name
    CODE:
    PUSHMARK(SP) ;
    call_pv(name, G_DISCARD|G_NOARGS) ;
```

That is fine as far as it goes. The thing is, the Perl subroutine can be specified as only a string. For Perl 4 this was adequate, but Perl 5 allows references to subroutines and anonymous subroutines. This is where *call_sv* is useful.

The code below for *CallSubSV* is identical to *CallSubPV* except that the name parameter is now defined as an SV* and we use *call_sv* instead of *call_pv*.

```
void
CallSubSV(name)
    SV *     name
    CODE:
    PUSHMARK(SP) ;
    call_sv(name, G_DISCARD|G_NOARGS) ;
```

Because we are using an SV to call *fred* the following can all be used

```
CallSubSV("fred") ;
CallSubSV(&fred) ;
$ref = &fred ;
CallSubSV($ref) ;
CallSubSV( sub { print "Hello there\n" } ) ;
```

As you can see, *call_sv* gives you much greater flexibility in how you can specify the Perl subroutine.

You should note that if it is necessary to store the SV (name in the example above) which corresponds to the Perl subroutine so that it can be used later in the program, it not enough just to store a copy of the pointer to the SV. Say the code above had been like this

```
static SV * rememberSub ;

void
SaveSub1(name)
    SV *     name
    CODE:
    rememberSub = name ;

void
CallSavedSub1()
    CODE:
    PUSHMARK(SP) ;
    call_sv(rememberSub, G_DISCARD|G_NOARGS) ;
```

The reason this is wrong is that by the time you come to use the pointer *rememberSub* in *CallSavedSub1*, it may or may not still refer to the Perl subroutine that was recorded in *SaveSub1*.

This is particularly true for these cases

```
SaveSub1(\&fred) ;
CallSavedSub1() ;

SaveSub1( sub { print "Hello there\n" } ) ;
CallSavedSub1() ;
```

By the time each of the `SaveSub1` statements above have been executed, the SV*s which corresponded to the parameters will no longer exist. Expect an error message from Perl of the form

```
Can't use an undefined value as a subroutine reference at ...
```

for each of the `CallSavedSub1` lines.

Similarly, with this code

```
$ref = \&fred ;
SaveSub1($ref) ;
$ref = 47 ;
CallSavedSub1() ;
```

you can expect one of these messages (which you actually get is dependent on the version of Perl you are using)

```
Not a CODE reference at ...
Undefined subroutine &main::47 called ...
```

The variable `$ref` may have referred to the subroutine `fred` whenever the call to `SaveSub1` was made but by the time `CallSavedSub1` gets called it now holds the number 47. Because we saved only a pointer to the original SV in `SaveSub1`, any changes to `$ref` will be tracked by the pointer `rememberSub`. This means that whenever `CallSavedSub1` gets called, it will attempt to execute the code which is referenced by the SV* `rememberSub`. In this case though, it now refers to the integer 47, so expect Perl to complain loudly.

A similar but more subtle problem is illustrated with this code

```
$ref = \&fred ;
SaveSub1($ref) ;
$ref = \&joe ;
CallSavedSub1() ;
```

This time whenever `CallSavedSub1` get called it will execute the Perl subroutine `joe` (assuming it exists) rather than `fred` as was originally requested in the call to `SaveSub1`.

To get around these problems it is necessary to take a full copy of the SV. The code below shows `SaveSub2` modified to do that

```
static SV * keepSub = (SV*)NULL ;

void
SaveSub2(name)
    SV *      name
    CODE:
    /* Take a copy of the callback */
    if (keepSub == (SV*)NULL)
        /* First time, so create a new SV */
        keepSub = newSVsv(name) ;
    else
        /* Been here before, so overwrite */
        SvSetSV(keepSub, name) ;
```

```
void
CallSavedSub2()
    CODE:
    PUSHMARK(SP) ;
    call_sv(keepSub, G_DISCARD|G_NOARGS) ;
```

To avoid creating a new SV every time `SaveSub2` is called, the function first checks to see if it has been called before. If not, then space for a new SV is allocated and the reference to the Perl subroutine, name is copied to the variable `keepSub` in one operation using `newSVsv`. Thereafter, whenever `SaveSub2` is called the existing SV, `keepSub`, is overwritten with the new value using `SvSetSV`.

Using `call_argv`

Here is a Perl subroutine which prints whatever parameters are passed to it.

```
sub PrintList
{
    my(@list) = @_ ;
    foreach (@list) { print "$_\n" }
}
```

and here is an example of `call_argv` which will call `PrintList`.

```
static char * words[] = {"alpha", "beta", "gamma", "delta", NULL} ;
static void
call_PrintList()
{
    dSP ;
    call_argv("PrintList", G_DISCARD, words) ;
}
```

Note that it is not necessary to call `PUSHMARK` in this instance. This is because `call_argv` will do it for you.

Using `call_method`

Consider the following Perl code

```
{
    package Mine ;

    sub new
    {
        my($type) = shift ;
        bless [ @_ ]
    }

    sub Display
    {
        my ($self, $index) = @_ ;
        print "$index: $$self[$index]\n" ;
    }

    sub PrintID
    {
        my($class) = @_ ;
        print "This is Class $class version 1.0\n" ;
    }
}
```

It implements just a very simple class to manage an array. Apart from the constructor, `new`, it declares methods, one static and one virtual. The static method, `PrintID`, prints out simply the class name and a

version number. The virtual method, `Display`, prints out a single element of the array. Here is an all Perl example of using it.

```
$a = new Mine ('red', 'green', 'blue') ;
$a->Display(1) ;
PrintID Mine;
```

will print

```
1: green
This is Class Mine version 1.0
```

Calling a Perl method from C is fairly straightforward. The following things are required

- a reference to the object for a virtual method or the name of the class for a static method.
- the name of the method.
- any other parameters specific to the method.

Here is a simple XSUB which illustrates the mechanics of calling both the `PrintID` and `Display` methods from C.

```
void
call_Method(ref, method, index)
    SV *      ref
    char *    method
    int       index
    CODE:
    PUSHMARK(SP);
    XPUSHs(ref);
    XPUSHs(sv_2mortal(newSViv(index))) ;
    PUTBACK;

    call_method(method, G_DISCARD) ;

void
call_PrintID(class, method)
    char *    class
    char *    method
    CODE:
    PUSHMARK(SP);
    XPUSHs(sv_2mortal(newSVpv(class, 0))) ;
    PUTBACK;

    call_method(method, G_DISCARD) ;
```

So the methods `PrintID` and `Display` can be invoked like this

```
$a = new Mine ('red', 'green', 'blue') ;
call_Method($a, 'Display', 1) ;
call_PrintID('Mine', 'PrintID') ;
```

The only thing to note is that in both the static and virtual methods, the method name is not passed via the stack—it is used as the first parameter to `call_method`.

Using GIMME_V

Here is a trivial XSUB which prints the context in which it is currently executing.

```
void
PrintContext()
    CODE:
    I32 gimme = GIMME_V;
```

```

    if (gimme == G_VOID)
        printf ("Context is Void\n") ;
    else if (gimme == G_SCALAR)
        printf ("Context is Scalar\n") ;
    else
        printf ("Context is Array\n") ;

```

and here is some Perl to test it

```

    PrintContext ;
    $a = PrintContext ;
    @a = PrintContext ;

```

The output from that will be

```

    Context is Void
    Context is Scalar
    Context is Array

```

Using Perl to dispose of temporaries

In the examples given to date, any temporaries created in the callback (i.e., parameters passed on the stack to the *call_** function or values returned via the stack) have been freed by one of these methods

- specifying the G_DISCARD flag with *call_**.
- explicitly disposed of using the ENTER/SAVETMPS – FREETMPS/LEAVE pairing.

There is another method which can be used, namely letting Perl do it for you automatically whenever it regains control after the callback has terminated. This is done by simply not using the

```

    ENTER ;
    SAVETMPS ;
    ...
    FREETMPS ;
    LEAVE ;

```

sequence in the callback (and not, of course, specifying the G_DISCARD flag).

If you are going to use this method you have to be aware of a possible memory leak which can arise under very specific circumstances. To explain these circumstances you need to know a bit about the flow of control between Perl and the callback routine.

The examples given at the start of the document (an error handler and an event driven program) are typical of the two main sorts of flow control that you are likely to encounter with callbacks. There is a very important distinction between them, so pay attention.

In the first example, an error handler, the flow of control could be as follows. You have created an interface to an external library. Control can reach the external library like this

```
perl --> XSUB --> external library
```

Whilst control is in the library, an error condition occurs. You have previously set up a Perl callback to handle this situation, so it will get executed. Once the callback has finished, control will drop back to Perl again. Here is what the flow of control will be like in that situation

```

perl --> XSUB --> external library
    ...
    error occurs
    ...
    external library --> call_* --> perl
perl <-- XSUB <-- external library <-- call_* <-----+

```


After processing of the error using `call_*` is completed, control reverts back to Perl more or less immediately.

In the diagram, the further right you go the more deeply nested the scope is. It is only when control is back with perl on the extreme left of the diagram that you will have dropped back to the enclosing scope and any temporaries you have left hanging around will be freed.

In the second example, an event driven program, the flow of control will be more like this

```
perl --> XSUB --> event handler
                        ...
                        event handler --> call_* --> perl
                        |
                        event handler <-- call_* <-----+
                        ...
                        event handler --> call_* --> perl
                        |
                        event handler <-- call_* <-----+
                        ...
                        event handler --> call_* --> perl
                        |
                        event handler <-- call_* <-----+
```

In this case the flow of control can consist of only the repeated sequence

```
event handler --> call_* --> perl
```

for practically the complete duration of the program. This means that control may *never* drop back to the surrounding scope in Perl at the extreme left.

So what is the big problem? Well, if you are expecting Perl to tidy up those temporaries for you, you might be in for a long wait. For Perl to dispose of your temporaries, control must drop back to the enclosing scope at some stage. In the event driven scenario that may never happen. This means that as time goes on, your program will create more and more temporaries, none of which will ever be freed. As each of these temporaries consumes some memory your program will eventually consume all the available memory in your system—kapow!

So here is the bottom line—if you are sure that control will revert back to the enclosing Perl scope fairly quickly after the end of your callback, then it isn't absolutely necessary to dispose explicitly of any temporaries you may have created. Mind you, if you are at all uncertain about what to do, it doesn't do any harm to tidy up anyway.

Strategies for storing Callback Context Information

Potentially one of the trickiest problems to overcome when designing a callback interface can be figuring out how to store the mapping between the C callback function and the Perl equivalent.

To help understand why this can be a real problem first consider how a callback is set up in an all C environment. Typically a C API will provide a function to register a callback. This will expect a pointer to a function as one of its parameters. Below is a call to a hypothetical function `register_fatal` which registers the C function to get called when a fatal error occurs.

```
register_fatal(cb1) ;
```

The single parameter `cb1` is a pointer to a function, so you must have defined `cb1` in your code, say something like this

```
static void
cb1()
{
    printf ("Fatal Error\n") ;
    exit(1) ;
}
```

```
}
```

Now change that to call a Perl subroutine instead

```
static SV * callback = (SV*)NULL;

static void
cb1()
{
    dSP ;

    PUSHMARK(SP) ;

    /* Call the Perl sub to process the callback */
    call_sv(callback, G_DISCARD) ;
}

void
register_fatal(fn)
    SV *      fn
    CODE:
    /* Remember the Perl sub */
    if (callback == (SV*)NULL)
        callback = newSVsv(fn) ;
    else
        SvSetSV(callback, fn) ;

    /* register the callback with the external library */
    register_fatal(cb1) ;
```

where the Perl equivalent of `register_fatal` and the callback it registers, `pcb1`, might look like this

```
# Register the sub pcb1
register_fatal(&pcb1) ;

sub pcb1
{
    die "I'm dying...\n" ;
}
```

The mapping between the C callback and the Perl equivalent is stored in the global variable `callback`.

This will be adequate if you ever need to have only one callback registered at any time. An example could be an error handler like the code sketched out above. Remember though, repeated calls to `register_fatal` will replace the previously registered callback function with the new one.

Say for example you want to interface to a library which allows asynchronous file i/o. In this case you may be able to register a callback whenever a read operation has completed. To be of any use we want to be able to call separate Perl subroutines for each file that is opened. As it stands, the error handler example above would not be adequate as it allows only a single callback to be defined at any time. What we require is a means of storing the mapping between the opened file and the Perl subroutine we want to be called for that file.

Say the i/o library has a function `asynch_read` which associates a C function `ProcessRead` with a file handle `fh`—this assumes that it has also provided some routine to open the file and so obtain the file handle.

```
asynch_read(fh, ProcessRead)
```

This may expect the C *ProcessRead* function of this form

```
void
ProcessRead(fh, buffer)
int fh ;
```

```

char*buffer ;
{
    ...
}

```

To provide a Perl interface to this library we need to be able to map between the `fh` parameter and the Perl subroutine we want called. A hash is a convenient mechanism for storing this mapping. The code below shows a possible implementation

```

static HV * Mapping = (HV*)NULL ;

void
asynch_read(fh, callback)
    int      fh
    SV *     callback
CODE:
/* If the hash doesn't already exist, create it */
if (Mapping == (HV*)NULL)
    Mapping = newHV() ;

/* Save the fh -> callback mapping */
hv_store(Mapping, (char*)&fh, sizeof(fh), newSVsv(callback), 0) ;

/* Register with the C Library */
asynch_read(fh, asynch_read_if) ;

```

and `asynch_read_if` could look like this

```

static void
asynch_read_if(fh, buffer)
    int fh ;
    char *      buffer ;
{
    dSP ;
    SV ** sv ;

    /* Get the callback associated with fh */
    sv = hv_fetch(Mapping, (char*)&fh , sizeof(fh), FALSE) ;
    if (sv == (SV**)NULL)
        croak("Internal error...\n") ;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(fh))) ;
    XPUSHs(sv_2mortal(newSVpv(buffer, 0))) ;
    PUTBACK ;

    /* Call the Perl sub */
    call_sv(*sv, G_DISCARD) ;
}

```

For completeness, here is `asynch_close`. This shows how to remove the entry from the hash `Mapping`.

```

void
asynch_close(fh)
    int      fh
CODE:
/* Remove the entry from the hash */
(void) hv_delete(Mapping, (char*)&fh, sizeof(fh), G_DISCARD) ;

/* Now call the real asynch_close */
asynch_close(fh) ;

```

So the Perl interface would look like this

```
sub callback1
{
    my($handle, $buffer) = @_ ;
}

# Register the Perl callback
asynch_read($fh, \&callback1) ;

asynch_close($fh) ;
```

The mapping between the C callback and Perl is stored in the global hash `Mapping` this time. Using a hash has the distinct advantage that it allows an unlimited number of callbacks to be registered.

What if the interface provided by the C callback doesn't contain a parameter which allows the file handle to Perl subroutine mapping? Say in the asynchronous i/o package, the callback function gets passed only the buffer parameter like this

```
void
ProcessRead(buffer)
char *      buffer ;
{
    ...
}
```

Without the file handle there is no straightforward way to map from the C callback to the Perl subroutine.

In this case a possible way around this problem is to predefine a series of C functions to act as the interface to Perl, thus

```
#define MAX_CB          3
#define NULL_HANDLE -1
typedef void (*FnMap) () ;

struct MapStruct {
    FnMap    Function ;
    SV *     PerlSub ;
    int      Handle ;
} ;

static void fn1() ;
static void fn2() ;
static void fn3() ;

static struct MapStruct Map [MAX_CB] =
{
    { fn1, NULL, NULL_HANDLE },
    { fn2, NULL, NULL_HANDLE },
    { fn3, NULL, NULL_HANDLE }
} ;

static void
Pcb(index, buffer)
int index ;
char * buffer ;
{
    dSP ;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSVpv(buffer, 0))) ;
```

```

        PUTBACK ;

        /* Call the Perl sub */
        call_sv(Map[index].PerlSub, G_DISCARD) ;
    }

    static void
    fn1(buffer)
    char * buffer ;
    {
        Pcb(0, buffer) ;
    }

    static void
    fn2(buffer)
    char * buffer ;
    {
        Pcb(1, buffer) ;
    }

    static void
    fn3(buffer)
    char * buffer ;
    {
        Pcb(2, buffer) ;
    }

    void
    array_asynch_read(fh, callback)
        int      fh
        SV *      callback
    CODE:
        int index ;
        int null_index = MAX_CB ;

        /* Find the same handle or an empty entry */
        for (index = 0 ; index < MAX_CB ; ++index)
        {
            if (Map[index].Handle == fh)
                break ;

            if (Map[index].Handle == NULL_HANDLE)
                null_index = index ;
        }

        if (index == MAX_CB && null_index == MAX_CB)
            croak ("Too many callback functions registered\n") ;

        if (index == MAX_CB)
            index = null_index ;

        /* Save the file handle */
        Map[index].Handle = fh ;

        /* Remember the Perl sub */
        if (Map[index].PerlSub == (SV*)NULL)
            Map[index].PerlSub = newSVsv(callback) ;
        else
            SvSetSV(Map[index].PerlSub, callback) ;

```

```

    asynch_read(fh, Map[index].Function) ;

void
array_asynch_close(fh)
    int      fh
    CODE:
    int index ;

    /* Find the file handle */
    for (index = 0; index < MAX_CB ; ++ index)
        if (Map[index].Handle == fh)
            break ;

    if (index == MAX_CB)
        croak ("could not close fh %d\n", fh) ;

    Map[index].Handle = NULL_HANDLE ;
    SvREFCNT_dec (Map[index].PerlSub) ;
    Map[index].PerlSub = (SV*)NULL ;

    asynch_close(fh) ;

```

In this case the functions `fn1`, `fn2`, and `fn3` are used to remember the Perl subroutine to be called. Each of the functions holds a separate hard-wired index which is used in the function `Pcb` to access the `Map` array and actually call the Perl subroutine.

There are some obvious disadvantages with this technique.

Firstly, the code is considerably more complex than with the previous example.

Secondly, there is a hard-wired limit (in this case 3) to the number of callbacks that can exist simultaneously. The only way to increase the limit is by modifying the code to add more functions and then recompiling. None the less, as long as the number of functions is chosen with some care, it is still a workable solution and in some cases is the only one available.

To summarize, here are a number of possible methods for you to consider for storing the mapping between C and the Perl callback

1. Ignore the problem – Allow only 1 callback

For a lot of situations, like interfacing to an error handler, this may be a perfectly adequate solution.

2. Create a sequence of callbacks – hard wired limit

If it is impossible to tell from the parameters passed back from the C callback what the context is, then you may need to create a sequence of C callback interface functions, and store pointers to each in an array.

3. Use a parameter to map to the Perl callback

A hash is an ideal mechanism to store the mapping between C and Perl.

Alternate Stack Manipulation

Although I have made use of only the `POP*` macros to access values returned from Perl subroutines, it is also possible to bypass these macros and read the stack using the `ST` macro (See [perlxs](#) for a full description of the `ST` macro).

Most of the time the `POP*` macros should be adequate, the main problem with them is that they force you to process the returned values in sequence. This may not be the most suitable way to process the values in some cases. What we want is to be able to access the stack in a random order. The `ST` macro as used when coding an `XSUB` is ideal for this purpose.

The code below is the example given in the section *Returning a list of values* recoded to use `ST` instead of `POP*`.

```

static void
call_AddSubtract2(a, b)
int a ;
int b ;
{
    dSP ;
    I32 ax ;
    int count ;

    ENTER ;
    SAVETMPS;

    PUSHMARK(SP) ;
    XPUSHs(sv_2mortal(newSViv(a)));
    XPUSHs(sv_2mortal(newSViv(b)));
    PUTBACK ;

    count = call_pv("AddSubtract", G_ARRAY);

    SPAGAIN ;
    SP -= count ;
    ax = (SP - PL_stack_base) + 1 ;

    if (count != 2)
        croak("Big trouble\n") ;

    printf ("%d + %d = %d\n", a, b, SvIV(ST(0))) ;
    printf ("%d - %d = %d\n", a, b, SvIV(ST(1))) ;

    PUTBACK ;
    FREETMPS ;
    LEAVE ;
}

```

Notes

1. Notice that it was necessary to define the variable `ax`. This is because the `ST` macro expects it to exist. If we were in an `XSUB` it would not be necessary to define `ax` as it is already defined for you.
2. The code

```

    SPAGAIN ;
    SP -= count ;
    ax = (SP - PL_stack_base) + 1 ;

```

sets the stack up so that we can use the `ST` macro.

3. Unlike the original coding of this example, the returned values are not accessed in reverse order. So `ST(0)` refers to the first value returned by the Perl subroutine and `ST(count-1)` refers to the last.

Creating and calling an anonymous subroutine in C

As we've already shown, `call_sv` can be used to invoke an anonymous subroutine. However, our example showed a Perl script invoking an `XSUB` to perform this operation. Let's see how it can be done inside our C code:

```

...
SV *cvrv = eval_pv("sub { print 'You will not find me cluttering any namespace!' }",
...
call_sv(cvrv, G_VOID|G_NOARGS);

```

`eval_pv` is used to compile the anonymous subroutine, which will be the return value as well (read more about `eval_pv` in [eval_pv](#)). Once this code reference is in hand, it can be mixed in with all the previous examples we've shown.

SEE ALSO

[perlxs](#), [perlguts](#), [perlembed](#)

AUTHOR

Paul Marquess

Special thanks to the following people who assisted in the creation of the document.

Jeff Okamoto, Tim Bunce, Nick Gianniotis, Steve Kelem, Gurusamy Sarathy and Larry Wall.

DATE

Version 1.3, 14th Apr 1997

NAME

perlclib – Internal replacements for standard C library functions

DESCRIPTION

One thing Perl porters should note is that *perl* doesn't tend to use that much of the C standard library internally; you'll see very little use of, for example, the *ctype.h* functions in there. This is because Perl tends to reimplement or abstract standard library functions, so that we know exactly how they're going to operate.

This is a reference card for people who are familiar with the C library and who want to do things the Perl way; to tell them which functions they ought to use instead of the more normal C functions.

Conventions

In the following tables:

`t` is a type.

`p` is a pointer.

`n` is a number.

`s` is a string.

`sv`, `av`, `hv`, etc. represent variables of their respective types.

File Operations

Instead of the *stdio.h* functions, you should use the Perl abstraction layer. Instead of `FILE*` types, you need to be handling `PerlIO*` types. Don't forget that with the new PerlIO layered I/O abstraction `FILE*` types may not even be available. See also the `perlapi` documentation for more information about the following functions:

Instead Of:	Use:
<code>stdin</code>	<code>PerlIO_stdin()</code>
<code>stdout</code>	<code>PerlIO_stdout()</code>
<code>stderr</code>	<code>PerlIO_stderr()</code>
<code>fopen(fn, mode)</code>	<code>PerlIO_open(fn, mode)</code>
<code>freopen(fn, mode, stream)</code>	<code>PerlIO_reopen(fn, mode, perlio)</code> (Deprecated)
<code>fflush(stream)</code>	<code>PerlIO_flush(perlio)</code>
<code>fclose(stream)</code>	<code>PerlIO_close(perlio)</code>

File Input and Output

Instead Of:	Use:
<code>fprintf(stream, fmt, ...)</code>	<code>PerlIO_printf(perlio, fmt, ...)</code>
<code>[f]getc(stream)</code>	<code>PerlIO_getc(perlio)</code>
<code>[f]putc(stream, n)</code>	<code>PerlIO_putc(perlio, n)</code>
<code>ungetc(n, stream)</code>	<code>PerlIO_ungetc(perlio, n)</code>

Note that the PerlIO equivalents of `fread` and `fwrite` are slightly different from their C library counterparts:

<code>fread(p, size, n, stream)</code>	<code>PerlIO_read(perlio, buf, numbytes)</code>
<code>fwrite(p, size, n, stream)</code>	<code>PerlIO_write(perlio, buf, numbytes)</code>
<code>fputs(s, stream)</code>	<code>PerlIO_puts(perlio, s)</code>

There is no equivalent to `fgets`; one should use `sv_gets` instead:

<code>fgets(s, n, stream)</code>	<code>sv_gets(sv, perlio, append)</code>
----------------------------------	--

File Positioning

Instead Of:	Use:
<code>feof(stream)</code>	<code>PerlIO_eof(perlio)</code>
<code>fseek(stream, n, whence)</code>	<code>PerlIO_seek(perlio, n, whence)</code>
<code>rewind(stream)</code>	<code>PerlIO_rewind(perlio)</code>
<code>fgetpos(stream, p)</code>	<code>PerlIO_getpos(perlio, sv)</code>
<code>fsetpos(stream, p)</code>	<code>PerlIO_setpos(perlio, sv)</code>
<code>ferror(stream)</code>	<code>PerlIO_error(perlio)</code>
<code>clearerr(stream)</code>	<code>PerlIO_clearerr(perlio)</code>

Memory Management and String Handling

Instead Of:	Use:
<code>t* p = malloc(n)</code>	<code>New(id, p, n, t)</code>
<code>t* p = calloc(n, s)</code>	<code>Newz(id, p, n, t)</code>
<code>p = realloc(p, n)</code>	<code>Renew(p, n, t)</code>
<code>memcpy(dst, src, n)</code>	<code>Copy(src, dst, n, t)</code>
<code>memmove(dst, src, n)</code>	<code>Move(src, dst, n, t)</code>
<code>memcpy/*(struct foo *)</code>	<code>StructCopy(src, dst, t)</code>
<code>memset(dst, 0, n * sizeof(t))</code>	<code>Zero(dst, n, t)</code>
<code>memzero(dst, 0)</code>	<code>Zero(dst, n, char)</code>
<code>free(p)</code>	<code>Safefree(p)</code>
<code>strdup(p)</code>	<code>savepv(p)</code>
<code>strndup(p, n)</code>	<code>savepvn(p, n) (Hey, strndup doesn't exist!)</code>
<code>strstr(big, little)</code>	<code>instr(big, little)</code>
<code>strcmp(s1, s2)</code>	<code>strLE(s1, s2) / strEQ(s1, s2) / strGT(s1, s2)</code>
<code>strncmp(s1, s2, n)</code>	<code>strnNE(s1, s2, n) / strnEQ(s1, s2, n)</code>

Notice the different order of arguments to `Copy` and `Move` than used in `memcpy` and `memmove`.

Most of the time, though, you'll want to be dealing with SVs internally instead of raw `char *` strings:

<code>strlen(s)</code>	<code>sv_len(sv)</code>
<code>strcpy(dt, src)</code>	<code>sv_setpv(sv, s)</code>
<code>strncpy(dt, src, n)</code>	<code>sv_setpvn(sv, s, n)</code>
<code>strcat(dt, src)</code>	<code>sv_catpv(sv, s)</code>
<code>strncat(dt, src)</code>	<code>sv_catpvn(sv, s)</code>
<code>sprintf(s, fmt, ...)</code>	<code>sv_setpvf(sv, fmt, ...)</code>

Note also the existence of `sv_catpvf` and `sv_vcatpvfn`, combining concatenation with formatting.

Sometimes instead of zeroing the allocated heap by using `Newz()` you should consider "poisoning" the data. This means writing a bit pattern into it that should be illegal as pointers (and floating point numbers), and also hopefully surprising enough as integers, so that any code attempting to use the data without forethought will break sooner rather than later. Poisoning can be done using the `Poison()` macro, which has similar arguments as `Zero()`:

```
Poison(dst, n, t)
```

Character Class Tests

There are two types of character class tests that Perl implements: one type deals in chars and are thus **not** Unicode aware (and hence deprecated unless you **know** you should use them) and the other type deal in UVs and know about Unicode properties. In the following table, `c` is a char, and `u` is a Unicode codepoint.

Instead Of:	Use:	But better use:
-------------	------	-----------------

isalnum(c)	isALNUM(c)	isALNUM_uni(u)
isalpha(c)	isALPHA(c)	isALPHA_uni(u)
iscntrl(c)	isCNTRL(c)	isCNTRL_uni(u)
isdigit(c)	isDIGIT(c)	isDIGIT_uni(u)
isgraph(c)	isGRAPH(c)	isGRAPH_uni(u)
islower(c)	isLOWER(c)	isLOWER_uni(u)
isprint(c)	isPRINT(c)	isPRINT_uni(u)
ispunct(c)	isPUNCT(c)	isPUNCT_uni(u)
isspace(c)	isSPACE(c)	isSPACE_uni(u)
isupper(c)	isUPPER(c)	isUPPER_uni(u)
isxdigit(c)	isXDIGIT(c)	isXDIGIT_uni(u)
tolower(c)	toLOWER(c)	toLOWER_uni(u)
toupper(c)	toUPPER(c)	toUPPER_uni(u)

stdlib.h functions

Instead Of:	Use:
atof(s)	Atof(s)
atol(s)	Atol(s)
strtod(s, *p)	Nothing. Just don't use it.
strtol(s, *p, n)	Strtol(s, *p, n)
strtoul(s, *p, n)	Strtoul(s, *p, n)

Notice also the `grok_bin`, `grok_hex`, and `grok_oct` functions in *numeric.c* for converting strings representing numbers in the respective bases into NVs.

In theory `Strtol` and `Strtoul` may not be defined if the machine perl is built on doesn't actually have `strtol` and `strtoul`. But as those 2 functions are part of the 1989 ANSI C spec we suspect you'll find them everywhere by now.

int rand()	double Drand01()
srand(n)	{ seedDrand01((Rand_seed_t)n); PL_srand_called = TRUE; }
exit(n)	my_exit(n)
system(s)	Don't. Look at <code>pp_system</code> or use <code>my_popen</code>
getenv(s)	PerlEnv_getenv(s)
setenv(s, val)	my_putenv(s, val)

Miscellaneous functions

You should not even **want** to use *setjmp.h* functions, but if you think you do, use the `JMPENV` stack in *scope.h* instead.

For signal/sigaction, use `rsignal(signo, handler)`.

SEE ALSO

`perlapi`, `perlapiio`, `perlguts`

NAME

perlcompile – Introduction to the Perl Compiler–Translator

DESCRIPTION

Perl has always had a compiler: your source is compiled into an internal form (a parse tree) which is then optimized before being run. Since version 5.005, Perl has shipped with a module capable of inspecting the optimized parse tree (`B`), and this has been used to write many useful utilities, including a module that lets you turn your Perl into C source code that can be compiled into a native executable.

The `B` module provides access to the parse tree, and other modules ("back ends") do things with the tree. Some write it out as bytecode, C source code, or a semi-human-readable text. Another traverses the parse tree to build a cross-reference of which subroutines, formats, and variables are used where. Another checks your code for dubious constructs. Yet another back end dumps the parse tree back out as Perl source, acting as a source code beautifier or deobfuscator.

Because its original purpose was to be a way to produce C code corresponding to a Perl program, and in turn a native executable, the `B` module and its associated back ends are known as "the compiler", even though they don't really compile anything. Different parts of the compiler are more accurately a "translator", or an "inspector", but people want Perl to have a "compiler option" not an "inspector gadget". What can you do?

This document covers the use of the Perl compiler: which modules it comprises, how to use the most important of the back end modules, what problems there are, and how to work around them.

Layout

The compiler back ends are in the `B::` hierarchy, and the front-end (the module that you, the user of the compiler, will sometimes interact with) is the `O` module. Some back ends (e.g., `B::C`) have programs (e.g., *perlcc*) to hide the modules' complexity.

Here are the important back ends to know about, with their status expressed as a number from 0 (outline for later implementation) to 10 (if there's a bug in it, we're very surprised):

B::Bytecode

Stores the parse tree in a machine-independent format, suitable for later reloading through the `ByteLoader` module. Status: 5 (some things work, some things don't, some things are untested).

B::C

Creates a C source file containing code to rebuild the parse tree and resume the interpreter. Status: 6 (many things work adequately, including programs using Tk).

B::CC

Creates a C source file corresponding to the run time code path in the parse tree. This is the closest to a Perl-to-C translator there is, but the code it generates is almost incomprehensible because it translates the parse tree into a giant switch structure that manipulates Perl structures. Eventual goal is to reduce (given sufficient type information in the Perl program) some of the Perl data structure manipulations into manipulations of C-level ints, floats, etc. Status: 5 (some things work, including uncomplicated Tk examples).

B::Lint

Complains if it finds dubious constructs in your source code. Status: 6 (it works adequately, but only has a very limited number of areas that it checks).

B::Deparse

Recreates the Perl source, making an attempt to format it coherently. Status: 8 (it works nicely, but a few obscure things are missing).

B::Xref

Reports on the declaration and use of subroutines and variables. Status: 8 (it works nicely, but still has a few lingering bugs).

Using The Back Ends

The following sections describe how to use the various compiler back ends. They're presented roughly in order of maturity, so that the most stable and proven back ends are described first, and the most experimental and incomplete back ends are described last.

The O module automatically enabled the `-c` flag to Perl, which prevents Perl from executing your code once it has been compiled. This is why all the back ends print:

```
myperlprogram syntax OK
```

before producing any other output.

The Cross Referencing Back End

The cross referencing back end (B::Xref) produces a report on your program, breaking down declarations and uses of subroutines and variables (and formats) by file and subroutine. For instance, here's part of the report from the *pod2man* program that comes with Perl:

```
Subroutine clear_noremap
Package (lexical)
  $ready_to_print    i1069, 1079
Package main
  $&                  1086
  $.                  1086
  $0                  1086
  $1                  1087
  $2                  1085, 1085
  $3                  1085, 1085
  $ARGV               1086
  %HTML_Escapes       1085, 1085
```

This shows the variables used in the subroutine `clear_noremap`. The variable `$ready_to_print` is a `my()` (lexical) variable, introduced (first declared with `my()`) on line 1069, and used on line 1079. The variable `$&` from the main package is used on 1086, and so on.

A line number may be prefixed by a single letter:

- i Lexical variable introduced (declared with `my()`) for the first time.
- & Subroutine or method call.
- s Subroutine defined.
- r Format defined.

The most useful option the cross referencer has is to save the report to a separate file. For instance, to save the report on *myperlprogram* to the file *report*:

```
$ perl -MO=Xref,-oreport myperlprogram
```

The Decompiling Back End

The Deparse back end turns your Perl source back into Perl source. It can reformat along the way, making it useful as a de-obfuscator. The most basic way to use it is:

```
$ perl -MO=Deparse myperlprogram
```

You'll notice immediately that Perl has no idea of how to paragraph your code. You'll have to separate chunks of code from each other with newlines by hand. However, watch what it will do with one-liners:

```
$ perl -MO=Deparse -e '$op=shift||die "usage: $0
code [...]";chomp(@ARGV=<>)unless@ARGV; for(@ARGV){$was=$_;eval$op;
die$@ if$@; rename$was,$_ unless$was eq $_}'
-e syntax OK
```

```

$op = shift @ARGV || die("usage: $0 code [...]");
chomp(@ARGV = <ARGV>) unless @ARGV;
foreach $_ (@ARGV) {
    $was = $_;
    eval $op;
    die "$@" if $@;
    rename $was, $_ unless $was eq $_;
}

```

The decompiler has several options for the code it generates. For instance, you can set the size of each indent from 4 (as above) to 2 with:

```
$ perl -MO=Deparse,-si2 myperlprogram
```

The **-p** option adds parentheses where normally they are omitted:

```

$ perl -MO=Deparse -e 'print "Hello, world\n"'
-e syntax OK
print "Hello, world\n";
$ perl -MO=Deparse,-p -e 'print "Hello, world\n"'
-e syntax OK
print("Hello, world\n");

```

See [B::Deparse](#) for more information on the formatting options.

The Lint Back End

The lint back end ([B::Lint](#)) inspects programs for poor style. One programmer's bad style is another programmer's useful tool, so options let you select what is complained about.

To run the style checker across your source code:

```
$ perl -MO=Lint myperlprogram
```

To disable context checks and undefined subroutines:

```
$ perl -MO=Lint,-context,-undefined-subs myperlprogram
```

See [B::Lint](#) for information on the options.

The Simple C Back End

This module saves the internal compiled state of your Perl program to a C source file, which can be turned into a native executable for that particular platform using a C compiler. The resulting program links against the Perl interpreter library, so it will not save you disk space (unless you build Perl with a shared library) or program size. It may, however, save you startup time.

The `perlcc` tool generates such executables by default.

```
perlcc myperlprogram.pl
```

The Bytecode Back End

This back end is only useful if you also have a way to load and execute the bytecode that it produces. The `ByteLoader` module provides this functionality.

To turn a Perl program into executable byte code, you can use `perlcc` with the **-b** switch:

```
perlcc -b myperlprogram.pl
```

The byte code is machine independent, so once you have a compiled module or program, it is as portable as Perl source (assuming that the user of the module or program has a modern-enough Perl interpreter to decode the byte code).

See **B::Bytecode** for information on options to control the optimization and nature of the code generated by the Bytecode module.

The Optimized C Back End

The optimized C back end will turn your Perl program's run time code-path into an equivalent (but optimized) C program that manipulates the Perl data structures directly. The program will still link against the Perl interpreter library, to allow for `eval()`, `s///e`, `require`, etc.

The `perlcc` tool generates such executables when using the `-opt` switch. To compile a Perl program (ending in `.pl` or `.p`):

```
perlcc -opt myperlprogram.pl
```

To produce a shared library from a Perl module (ending in `.pm`):

```
perlcc -opt Myperlmodule.pm
```

For more information, see [perlcc](#) and [B::CC](#).

Module List for the Compiler Suite

- B** This module is the introspective ("reflective" in Java terms) module, which allows a Perl program to inspect its innards. The back end modules all use this module to gain access to the compiled parse tree. You, the user of a back end module, will not need to interact with B.
- O** This module is the front-end to the compiler's back ends. Normally called something like this:

```
$ perl -MO=Deparse myperlprogram
```

This is like saying use `O 'Deparse'` in your Perl program.

B::Asmdata

This module is used by the B::Assembler module, which is in turn used by the B::Bytecode module, which stores a parse-tree as bytecode for later loading. It's not a back end itself, but rather a component of a back end.

B::Assembler

This module turns a parse-tree into data suitable for storing and later decoding back into a parse-tree. It's not a back end itself, but rather a component of a back end. It's used by the *assemble* program that produces bytecode.

B::Bblock

This module is used by the B::CC back end. It walks "basic blocks". A basic block is a series of operations which is known to execute from start to finish, with no possibility of branching or halting.

B::Bytecode

This module is a back end that generates bytecode from a program's parse tree. This bytecode is written to a file, from where it can later be reconstructed back into a parse tree. The goal is to do the expensive program compilation once, save the interpreter's state into a file, and then restore the state from the file when the program is to be executed. See for details about usage.

B::C

This module writes out C code corresponding to the parse tree and other interpreter internal structures. You compile the corresponding C file, and get an executable file that will restore the internal structures and the Perl interpreter will begin running the program. See for details about usage.

B::CC

This module writes out C code corresponding to your program's operations. Unlike the B::C module, which merely stores the interpreter and its state in a C program, the B::CC module makes a C program that does not involve the interpreter. As a consequence, programs translated into C by B::CC can execute faster than normal interpreted programs. See for details about usage.

B::Concise

This module prints a concise (but complete) version of the Perl parse tree. Its output is more customizable than the one of B::Terse or B::Debug (and it can emulate them). This module useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

B::Debug

This module dumps the Perl parse tree in verbose detail to STDOUT. It's useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

B::Deparse

This module produces Perl source code from the compiled parse tree. It is useful in debugging and deconstructing other people's code, also as a pretty-printer for your own source. See for details about usage.

B::Disassembler

This module turns bytecode back into a parse tree. It's not a back end itself, but rather a component of a back end. It's used by the *disassemble* program that comes with the bytecode.

B::Lint

This module inspects the compiled form of your source code for things which, while some people frown on them, aren't necessarily bad enough to justify a warning. For instance, use of an array in scalar context without explicitly saying `scalar(@array)` is something that Lint can identify. See for details about usage.

B::Showlex

This module prints out the `my()` variables used in a function or a file. To get a list of the `my()` variables used in the subroutine `mysub()` defined in the file `myperlprogram`:

```
$ perl -MO=Showlex,mysub myperlprogram
```

To get a list of the `my()` variables used in the file `myperlprogram`:

```
$ perl -MO=Showlex myperlprogram
```

[BROKEN]

B::Stackobj

This module is used by the B::CC module. It's not a back end itself, but rather a component of a back end.

B::Stash

This module is used by the *perlcc* program, which compiles a module into an executable. B::Stash prints the symbol tables in use by a program, and is used to prevent B::CC from producing C code for the B::* and O modules. It's not a back end itself, but rather a component of a back end.

B::Terse

This module prints the contents of the parse tree, but without as much information as B::Debug. For comparison, `print "Hello, world."` produced 96 lines of output from B::Debug, but only 6 from B::Terse.

This module is useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

B::Xref

This module prints a report on where the variables, subroutines, and formats are defined and used within a program and the modules it loads. See for details about usage.

KNOWN PROBLEMS

The simple C backend currently only saves typeglobs with alphanumeric names.

The optimized C backend outputs code for more modules than it should (e.g., `DirHandle`). It also has little hope of properly handling `goto LABEL` outside the running subroutine (`goto &sub` is okay). `goto LABEL` currently does not work at all in this backend. It also creates a huge initialization function that gives C compilers headaches. Splitting the initialization function gives better results. Other problems include: unsigned math does not work correctly; some opcodes are handled incorrectly by default opcode handling mechanism.

`BEGIN{ }` blocks are executed while compiling your code. Any external state that is initialized in `BEGIN{ }`, such as opening files, initiating database connections etc., do not behave properly. To work around this, Perl has an `INIT{ }` block that corresponds to code being executed before your program begins running but after your program has finished being compiled. Execution order: `BEGIN{ }`, (possible save of state through compiler back-end), `INIT{ }`, program runs, `END{ }`.

AUTHOR

This document was originally written by Nathan Torkington, and is now maintained by the perl5-porters mailing list perl5-porters@perl.org.

NAME

perldata – Perl data types

DESCRIPTION**Variable names**

Perl has three built-in data types: scalars, arrays of scalars, and associative arrays of scalars, known as "hashes". Normal arrays are ordered lists of scalars indexed by number, starting with 0 and with negative subscripts counting from the end. Hashes are unordered collections of scalar values indexed by their associated string key.

Values are usually referred to by name, or through a named reference. The first character of the name tells you to what sort of data structure it refers. The rest of the name tells you the particular value to which it refers. Usually this name is a single *identifier*, that is, a string beginning with a letter or underscore, and containing letters, underscores, and digits. In some cases, it may be a chain of identifiers, separated by `::` (or by the slightly archaic `'`); all but the last are interpreted as names of packages, to locate the namespace in which to look up the final identifier (see [Packages](#) for details). It's possible to substitute for a simple identifier, an expression that produces a reference to the value at runtime. This is described in more detail below and in [perlref](#).

Perl also has its own built-in variables whose names don't follow these rules. They have strange names so they don't accidentally collide with one of your normal variables. Strings that match parenthesized parts of a regular expression are saved under names containing only digits after the `$` (see [perllop](#) and [perlre](#)). In addition, several special variables that provide windows into the inner working of Perl have names containing punctuation characters and control characters. These are documented in [perlvar](#).

Scalar values are always named with `$`, even when referring to a scalar that is part of an array or a hash. The `$` symbol works semantically like the English word "the" in that it indicates a single value is expected.

```
$days          # the simple scalar value "days"
$days[28]      # the 29th element of array @days
$days{'Feb'}   # the 'Feb' value from hash %days
$#days         # the last index of array @days
```

Entire arrays (and slices of arrays and hashes) are denoted by `@`, which works much like the word "these" or "those" does in English, in that it indicates multiple values are expected.

```
@days          # ($days[0], $days[1], ... $days[n])
@days[3,4,5]    # same as ($days[3], $days[4], $days[5])
@days{'a','c'}  # same as ($days{'a'}, $days{'c'})
```

Entire hashes are denoted by `%`:

```
%days          # (key1, val1, key2, val2 ...)
```

In addition, subroutines are named with an initial `&`, though this is optional when unambiguous, just as the word "do" is often redundant in English. Symbol table entries can be named with an initial `*`, but you don't really care about that yet (if ever :-).

Every variable type has its own namespace, as do several non-variable identifiers. This means that you can, without fear of conflict, use the same name for a scalar variable, an array, or a hash—or, for that matter, for a filehandle, a directory handle, a subroutine name, a format name, or a label. This means that `$foo` and `@foo` are two different variables. It also means that `$foo[1]` is a part of `@foo`, not a part of `$foo`. This may seem a bit weird, but that's okay, because it is weird.

Because variable references always start with `$`, `@`, or `%`, the "reserved" words aren't in fact reserved with respect to variable names. They *are* reserved with respect to labels and filehandles, however, which don't have an initial special character. You can't have a filehandle named "log", for instance. Hint: you could say `open(LOG, 'logfile')` rather than `open(log, 'logfile')`. Using uppercase filehandles

also improves readability and protects you from conflict with future reserved words. Case *is* significant—"FOO", "Foo", and "foo" are all different names. Names that start with a letter or underscore may also contain digits and underscores.

It is possible to replace such an alphanumeric name with an expression that returns a reference to the appropriate type. For a description of this, see [perlref](#).

Names that start with a digit may contain only more digits. Names that do not start with a letter, underscore, digit or a caret (i.e. a control character) are limited to one character, e.g., `$%` or `$$`. (Most of these one character names have a predefined significance to Perl. For instance, `$$` is the current process id.)

Context

The interpretation of operations and values in Perl sometimes depends on the requirements of the context around the operation or value. There are two major contexts: list and scalar. Certain operations return list values in contexts wanting a list, and scalar values otherwise. If this is true of an operation it will be mentioned in the documentation for that operation. In other words, Perl overloads certain operations based on whether the expected return value is singular or plural. Some words in English work this way, like "fish" and "sheep".

In a reciprocal fashion, an operation provides either a scalar or a list context to each of its arguments. For example, if you say

```
int ( <STDIN> )
```

the integer operation provides scalar context for the `<` operator, which responds by reading one line from STDIN and passing it back to the integer operation, which will then find the integer value of that line and return that. If, on the other hand, you say

```
sort ( <STDIN> )
```

then the sort operation provides list context for `<`, which will proceed to read every line available up to the end of file, and pass that list of lines back to the sort routine, which will then sort those lines and return them as a list to whatever the context of the sort was.

Assignment is a little bit special in that it uses its left argument to determine the context for the right argument. Assignment to a scalar evaluates the right-hand side in scalar context, while assignment to an array or hash evaluates the righthand side in list context. Assignment to a list (or slice, which is just a list anyway) also evaluates the righthand side in list context.

When you use the `use warnings` pragma or Perl's `-w` command-line option, you may see warnings about useless uses of constants or functions in "void context". Void context just means the value has been discarded, such as a statement containing only `"fred";` or `getpwuid(0);`. It still counts as scalar context for functions that care whether or not they're being called in list context.

User-defined subroutines may choose to care whether they are being called in a void, scalar, or list context. Most subroutines do not need to bother, though. That's because both scalars and lists are automatically interpolated into lists. See [wantarray](#) for how you would dynamically discern your function's calling context.

Scalar values

All data in Perl is a scalar, an array of scalars, or a hash of scalars. A scalar may contain one single value in any of three different flavors: a number, a string, or a reference. In general, conversion from one form to another is transparent. Although a scalar may not directly hold multiple values, it may contain a reference to an array or hash which in turn contains multiple values.

Scalars aren't necessarily one thing or another. There's no place to declare a scalar variable to be of type "string", type "number", type "reference", or anything else. Because of the automatic conversion of scalars, operations that return scalars don't need to care (and in fact, cannot care) whether their caller is looking for a string, a number, or a reference. Perl is a contextually polymorphic language whose scalars can be strings, numbers, or references (which includes objects). Although strings and numbers are considered pretty much the same thing for nearly all purposes, references are strongly-typed, uncastable pointers with builtin

reference-counting and destructor invocation.

A scalar value is interpreted as TRUE in the Boolean sense if it is not the null string or the number 0 (or its string equivalent, "0"). The Boolean context is just a special kind of scalar context where no conversion to a string or a number is ever performed.

There are actually two varieties of null strings (sometimes referred to as "empty" strings), a defined one and an undefined one. The defined version is just a string of length zero, such as "". The undefined version is the value that indicates that there is no real value for something, such as when there was an error, or at end of file, or when you refer to an uninitialized variable or element of an array or hash. Although in early versions of Perl, an undefined scalar could become defined when first used in a place expecting a defined value, this no longer happens except for rare cases of autovivification as explained in [perlref](#). You can use the `defined()` operator to determine whether a scalar value is defined (this has no meaning on arrays or hashes), and the `undef()` operator to produce an undefined value.

To find out whether a given string is a valid non-zero number, it's sometimes enough to test it against both numeric 0 and also lexical "0" (although this will cause `-w` noises). That's because strings that aren't numbers count as 0, just as they do in `awk`:

```
if ($str == 0 && $str ne "0") {
    warn "That doesn't look like a number";
}
```

That method may be best because otherwise you won't treat IEEE notations like NaN or Infinity properly. At other times, you might prefer to determine whether string data can be used numerically by calling the `POSIX::strtod()` function or by inspecting your string with a regular expression (as documented in [perlre](#)).

```
warn "has nondigits"      if /\D/;
warn "not a natural number" unless /^-?\d+$/;          # rejects -3
warn "not an integer"     unless /^-?\d+$/;            # rejects +3
warn "not an integer"     unless /^[+-]?\d+$/;
warn "not a decimal number" unless /^-?\d+\.\d*$/;      # rejects .2
warn "not a decimal number" unless /^-?(?:\d+(?:\.\d*)?|\.\d+)$/;
warn "not a C float"
    unless /^[+-]?(?:\d|\.\d)\d*(\.\d*)?( [Ee] ([+-]?\d+) )?$/;
```

The length of an array is a scalar value. You may find the length of array `@days` by evaluating `$#days`, as in `csh`. However, this isn't the length of the array; it's the subscript of the last element, which is a different value since there is ordinarily a 0th element. Assigning to `$#days` actually changes the length of the array. Shortening an array this way destroys intervening values. Lengthening an array that was previously shortened does not recover values that were in those elements. (It used to do so in Perl 4, but we had to break this to make sure destructors were called when expected.)

You can also gain some minuscule measure of efficiency by pre-extending an array that is going to get big. You can also extend an array by assigning to an element that is off the end of the array. You can truncate an array down to nothing by assigning the null list `()` to it. The following are equivalent:

```
@whatever = ();
$#whatever = -1;
```

If you evaluate an array in scalar context, it returns the length of the array. (Note that this is not true of lists, which return the last value, like the C comma operator, nor of built-in functions, which return whatever they feel like returning.) The following is always true:

```
scalar(@whatever) == $#whatever - $[ + 1;
```

Version 5 of Perl changed the semantics of `$[:`: files that don't set the value of `$[` no longer need to worry about whether another file changed its value. (In other words, use of `$[` is deprecated.) So in general you can assume that

```
scalar(@whatever) == $#whatever + 1;
```

Some programmers choose to use an explicit conversion so as to leave nothing to doubt:

```
$element_count = scalar(@whatever);
```

If you evaluate a hash in scalar context, it returns false if the hash is empty. If there are any key/value pairs, it returns true; more precisely, the value returned is a string consisting of the number of used buckets and the number of allocated buckets, separated by a slash. This is pretty much useful only to find out whether Perl's internal hashing algorithm is performing poorly on your data set. For example, you stick 10,000 things in a hash, but evaluating %HASH in scalar context reveals "1/16", which means only one out of sixteen buckets has been touched, and presumably contains all 10,000 of your items. This isn't supposed to happen.

You can preallocate space for a hash by assigning to the `keys()` function. This rounds up the allocated buckets to the next power of two:

```
keys(%users) = 1000;           # allocate 1024 buckets
```

Scalar value constructors

Numeric literals are specified in any of the following floating point or integer formats:

```
12345
12345.67
.23E-10           # a very small number
3.14_15_92        # a very important number
4_294_967_296     # underscore for legibility
0xff              # hex
0xdead_beef       # more hex
0377              # octal
0b011011          # binary
```

You are allowed to use underscores (underbars) in numeric literals between digits for legibility. You could, for example, group binary digits by threes (as for a Unix-style mode argument such as 0b110_100_100) or by fours (to represent nibbles, as in 0b1010_0110) or in other groups.

String literals are usually delimited by either single or double quotes. They work much like quotes in the standard Unix shells: double-quoted string literals are subject to backslash and variable substitution; single-quoted strings are not (except for `\'` and `\\`). The usual C-style backslash rules apply for making characters such as newline, tab, etc., as well as some more exotic forms. See

[Quote and Quote-like Operators in perlop](#) for a list.

Hexadecimal, octal, or binary, representations in string literals (e.g. `'0xff'`) are not automatically converted to their integer representation. The `hex()` and `oct()` functions make these conversions for you. See [hex](#) and [oct](#) for more details.

You can also embed newlines directly in your strings, i.e., they can end on a different line than they begin. This is nice, but if you forget your trailing quote, the error will not be reported until Perl finds another line containing the quote character, which may be much further on in the script. Variable substitution inside strings is limited to scalar variables, arrays, and array or hash slices. (In other words, names beginning with `$` or `@`, followed by an optional bracketed expression as a subscript.) The following code segment prints out "The price is \$100."

```
$Price = '$100';    # not interpreted
print "The price is $Price.\n";    # interpreted
```

As in some shells, you can enclose the variable name in braces to disambiguate it from following alphanumeric (and underscores). You must also do this when interpolating a variable into a string to separate the variable name from a following double-colon or an apostrophe, since these would be otherwise treated as a package separator:

```
$who = "Larry";
```

```
print PASSWD "${who}::0:0:Superuser:/:/bin/perl\n";
print "We use ${who} speak when ${who}'s here.\n";
```

Without the braces, Perl would have looked for a `$whospeak`, a `$who::0`, and a `$who's` variable. The last two would be the `$0` and the `$s` variables in the (presumably) non-existent package `who`.

In fact, an identifier within such curlies is forced to be a string, as is any simple identifier within a hash subscript. Neither need quoting. Our earlier example, `$days{'Feb'}` can be written as `$days{Feb}` and the quotes will be assumed automatically. But anything more complicated in the subscript will be interpreted as an expression.

A literal of the form `v1.20.300.4000` is parsed as a string composed of characters with the specified ordinals. This form, known as *v-strings*, provides an alternative, more readable way to construct strings, rather than use the somewhat less readable interpolation form `"\x{1}\x{14}\x{12c}\x{fa0}"`. This is useful for representing Unicode strings, and for comparing version "numbers" using the string comparison operators, `cmp`, `gt`, `lt` etc. If there are two or more dots in the literal, the leading `v` may be omitted.

```
print v9786;                # prints UTF-8 encoded SMILEY, "\x{263a}"
print v102.111.111;        # prints "foo"
print 102.111.111;         # same
```

Such literals are accepted by both `require` and `use` for doing a version check. The `$_V` special variable also contains the running Perl interpreter's version in this form. See [\\$^V](#). Note that using the *v-strings* for IPv4 addresses is not portable unless you also use the `inet_aton()/inet_ntoa()` routines of the `Socket` package.

The special literals `__FILE__`, `__LINE__`, and `__PACKAGE__` represent the current filename, line number, and package name at that point in your program. They may be used only as separate tokens; they will not be interpolated into strings. If there is no current package (due to an empty package; directive), `__PACKAGE__` is the undefined value.

The two control characters `^D` and `^Z`, and the tokens `__END__` and `__DATA__` may be used to indicate the logical end of the script before the actual end of file. Any following text is ignored.

Text after `__DATA__` but may be read via the filehandle `PACKNAME::DATA`, where `PACKNAME` is the package that was current when the `__DATA__` token was encountered. The filehandle is left open pointing to the contents after `__DATA__`. It is the program's responsibility to `close DATA` when it is done reading from it. For compatibility with older scripts written before `__DATA__` was introduced, `__END__` behaves like `__DATA__` in the toplevel script (but not in files loaded with `require` or `do`) and leaves the remaining contents of the file accessible via `main::DATA`.

See [SelfLoader](#) for more description of `__DATA__`, and an example of its use. Note that you cannot read from the `DATA` filehandle in a `BEGIN` block: the `BEGIN` block is executed as soon as it is seen (during compilation), at which point the corresponding `__DATA__` (or `__END__`) token has not yet been seen.

A word that has no other interpretation in the grammar will be treated as if it were a quoted string. These are known as "barewords". As with filehandles and labels, a bareword that consists entirely of lowercase letters risks conflict with future reserved words, and if you use the `use warnings` pragma or the `-w` switch, Perl will warn you about any such words. Some people may wish to outlaw barewords entirely. If you say

```
use strict 'subs';
```

then any bareword that would NOT be interpreted as a subroutine call produces a compile-time error instead. The restriction lasts to the end of the enclosing block. An inner block may countermand this by saying `no strict 'subs'`.

Arrays and slices are interpolated into double-quoted strings by joining the elements with the delimiter specified in the `$"` variable (`$LIST_SEPARATOR` in English), space by default. The following are equivalent:

```
$temp = join("$", @ARGV);
```

```
system "echo $temp";

system "echo @ARGV";
```

Within search patterns (which also undergo double-quotish substitution) there is an unfortunate ambiguity: Is `/${foo}[bar]/` to be interpreted as `/${foo}[bar]/` (where `[bar]` is a character class for the regular expression) or as `/${foo}[bar]/` (where `[bar]` is the subscript to array `@foo`)? If `@foo` doesn't otherwise exist, then it's obviously a character class. If `@foo` exists, Perl takes a good guess about `[bar]`, and is almost always right. If it does guess wrong, or if you're just plain paranoid, you can force the correct interpretation with curly braces as above.

If you're looking for the information on how to use here-documents, which used to be here, that's been moved to [Quote and Quote-like Operators](#).

List value constructors

List values are denoted by separating individual values by commas (and enclosing the list in parentheses where precedence requires it):

```
(LIST)
```

In a context not requiring a list value, the value of what appears to be a list literal is simply the value of the final element, as with the C comma operator. For example,

```
@foo = ('cc', '-E', $bar);
```

assigns the entire list value to array `@foo`, but

```
$foo = ('cc', '-E', $bar);
```

assigns the value of variable `$bar` to the scalar variable `$foo`. Note that the value of an actual array in scalar context is the length of the array; the following assigns the value 3 to `$foo`:

```
@foo = ('cc', '-E', $bar);
$foo = @foo;           # $foo gets 3
```

You may have an optional comma before the closing parenthesis of a list literal, so that you can say:

```
@foo = (
    1,
    2,
    3,
);
```

To use a here-document to assign an array, one line per element, you might use an approach like this:

```
@sauces = <<End_Lines =~ m/(\S.*\S)/g;
    normal tomato
    spicy tomato
    green chile
    pesto
    white wine
End_Lines
```

LISTs do automatic interpolation of sublists. That is, when a LIST is evaluated, each element of the list is evaluated in list context, and the resulting list value is interpolated into LIST just as if each individual element were a member of LIST. Thus arrays and hashes lose their identity in a LIST—the list

```
(@foo, @bar, &SomeSub, %glarch)
```

contains all the elements of `@foo` followed by all the elements of `@bar`, followed by all the elements returned by the subroutine named `SomeSub` called in list context, followed by the key/value pairs of `%glarch`. To make a list reference that does *NOT* interpolate, see [perlref](#).

The null list is represented by `()`. Interpolating it in a list has no effect. Thus `((), (), ())` is equivalent to `()`. Similarly, interpolating an array with no elements is the same as if no array had been interpolated at that point.

This interpolation combines with the facts that the opening and closing parentheses are optional (except when necessary for precedence) and lists may end with an optional comma to mean that multiple commas within lists are legal syntax. The list `1,,3` is a concatenation of two lists, `1,` and `3`, the first of which ends with that optional comma. `1,,3` is `(1,),(3)` is `1,3` (And similarly for `1,,,3` is `(1,),(,),(,),3` is `1,3` and so on.) Not that we'd advise you to use this obfuscation.

A list value may also be subscripted like a normal array. You must put the list in parentheses to avoid ambiguity. For example:

```
# Stat returns list value.
$time = (stat($file))[8];

# SYNTAX ERROR HERE.
$time = stat($file)[8]; # OOPS, FORGOT PARENTHESES

# Find a hex digit.
$hexdigit = ('a','b','c','d','e','f')[$digit-10];

# A "reverse comma operator".
return (pop(@foo),pop(@foo))[0];
```

Lists may be assigned to only when each element of the list is itself legal to assign to:

```
($a, $b, $c) = (1, 2, 3);

($map{'red'}, $map{'blue'}, $map{'green'}) = (0x00f, 0x0f0, 0xf00);
```

An exception to this is that you may assign to `undef` in a list. This is useful for throwing away some of the return values of a function:

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

List assignment in scalar context returns the number of elements produced by the expression on the right side of the assignment:

```
$x = (($foo,$bar) = (3,2,1)); # set $x to 3, not 2
$x = (($foo,$bar) = f()); # set $x to f()'s return count
```

This is handy when you want to do a list assignment in a Boolean context, because most list functions return a null list when finished, which when assigned produces a 0, which is interpreted as FALSE.

It's also the source of a useful idiom for executing a function or performing an operation in list context and then counting the number of return values, by assigning to an empty list and then using that assignment in scalar context. For example, this code:

```
$count = () = $string =~ /\d+/g;
```

will place into `$count` the number of digit groups found in `$string`. This happens because the pattern match is in list context (since it is being assigned to the empty list), and will therefore return a list of all matching parts of the string. The list assignment in scalar context will translate that into the number of elements (here, the number of times the pattern matched) and assign that to `$count`. Note that simply using

```
$count = $string =~ /\d+/g;
```

would not have worked, since a pattern match in scalar context will only return true or false, rather than a count of matches.

The final element of a list assignment may be an array or a hash:


```
($a, $b, @rest) = split;
my($a, $b, %rest) = @_;
```

You can actually put an array or hash anywhere in the list, but the first one in the list will soak up all the values, and anything after it will become undefined. This may be useful in a `my()` or `local()`.

A hash can be initialized using a literal list holding pairs of items to be interpreted as a key and a value:

```
# same as map assignment above
%map = ('red', 0x00f, 'blue', 0x0f0, 'green', 0xf00);
```

While literal lists and named arrays are often interchangeable, that's not the case for hashes. Just because you can subscript a list value like a normal array does not mean that you can subscript a list value as a hash. Likewise, hashes included as parts of other lists (including parameters lists and return lists from functions) always flatten out into key/value pairs. That's why it's good to use references sometimes.

It is often more readable to use the `< =` operator between key/value pairs. The `< =` operator is mostly just a more visually distinctive synonym for a comma, but it also arranges for its left-hand operand to be interpreted as a string—if it's a bareword that would be a legal identifier. This makes it nice for initializing hashes:

```
%map = (
    red    => 0x00f,
    blue   => 0x0f0,
    green  => 0xf00,
);
```

or for initializing hash references to be used as records:

```
$rec = {
    witch => 'Mable the Merciless',
    cat   => 'Fluffy the Ferocious',
    date  => '10/31/1776',
};
```

or for using call-by-named-parameter to complicated functions:

```
$field = $query->radio_group(
    name      => 'group_name',
    values     => ['eenie', 'meenie', 'minie'],
    default    => 'meenie',
    linebreak  => 'true',
    labels     => \%labels
);
```

Note that just because a hash is initialized in that order doesn't mean that it comes out in that order. See [sort](#) for examples of how to arrange for an output ordering.

Slices

A common way to access an array or a hash is one scalar element at a time. You can also subscript a list to get a single element from it.

```
$whoami = $ENV{"USER"};           # one element from the hash
$parent = $ISA[0];                # one element from the array
$dir     = (getpwnam("daemon"))[7]; # likewise, but with list
```

A slice accesses several elements of a list, an array, or a hash simultaneously using a list of subscripts. It's more convenient than writing out the individual elements as a list of separate scalar values.

```
($him, $her) = @folks[0, -1];      # array slice
@them        = @folks[0 .. 3];     # array slice
($who, $home) = @ENV{"USER", "HOME"}; # hash slice
```

```
($uid, $dir) = (getpwnam("daemon"))[2,7]; # list slice
```

Since you can assign to a list of variables, you can also assign to an array or hash slice.

```
@days[3..5] = qw/Wed Thu Fri/;
@colors{'red','blue','green'}
           = (0xff0000, 0x0000ff, 0x00ff00);
@folks[0, -1] = @folks[-1, 0];
```

The previous assignments are exactly equivalent to

```
($days[3], $days[4], $days[5]) = qw/Wed Thu Fri/;
($colors{'red'}, $colors{'blue'}, $colors{'green'})
           = (0xff0000, 0x0000ff, 0x00ff00);
($folks[0], $folks[-1]) = ($folks[-1], $folks[0]);
```

Since changing a slice changes the original array or hash that it's slicing, a `foreach` construct will alter some—or even all—of the values of the array or hash.

```
foreach (@array[ 4 .. 10 ]) { s/peter/paul/ }

foreach (@hash{keys %hash}) {
    s/^\s+//;          # trim leading whitespace
    s/\s+$//;          # trim trailing whitespace
    s/(\w+)/\u\L$1/g;  # "titlecase" words
}
```

A slice of an empty list is still an empty list. Thus:

```
@a = () [1,0];          # @a has no elements
@b = (@a) [0,1];        # @b has no elements
@c = (0,1) [2,3];       # @c has no elements
```

But:

```
@a = (1) [1,0];         # @a has two elements
@b = (1,undef) [1,0,2]; # @b has three elements
```

This makes it easy to write loops that terminate when a null list is returned:

```
while ( ($home, $user) = (getpwent) [7,0]) {
    printf "%-8s %s\n", $user, $home;
}
```

As noted earlier in this document, the scalar sense of list assignment is the number of elements on the right-hand side of the assignment. The null list contains no elements, so when the password file is exhausted, the result is 0, not 2.

If you're confused about why you use an '@' there on a hash slice instead of a '%', think of it like this. The type of bracket (square or curly) governs whether it's an array or a hash being looked at. On the other hand, the leading symbol ('\$' or '@') on the array or hash indicates whether you are getting back a singular value (a scalar) or a plural one (a list).

Typeglobs and Filehandles

Perl uses an internal type called a *typeglob* to hold an entire symbol table entry. The type prefix of a typeglob is a *, because it represents all types. This used to be the preferred way to pass arrays and hashes by reference into a function, but now that we have real references, this is seldom needed.

The main use of typeglobs in modern Perl is create symbol table aliases. This assignment:

```
*this = *that;
```

makes \$this an alias for \$that, @this an alias for @that, %this an alias for %that, &this an alias for &that, etc. Much safer is to use a reference. This:

```
local *Here::blue = \ $There::green;
```

temporarily makes `$Here::blue` an alias for `$There::green`, but doesn't make `@Here::blue` an alias for `@There::green`, or `%Here::blue` an alias for `%There::green`, etc. See [Symbol Tables in perlmod](#) for more examples of this. Strange though this may seem, this is the basis for the whole module import/export system.

Another use for typeglobs is to pass filehandles into a function or to create new filehandles. If you need to use a typeglob to save away a filehandle, do it this way:

```
$fh = *STDOUT;
```

or perhaps as a real reference, like this:

```
$fh = \*STDOUT;
```

See [perlsub](#) for examples of using these as indirect filehandles in functions.

Typeglobs are also a way to create a local filehandle using the `local()` operator. These last until their block is exited, but may be passed back. For example:

```
sub newopen {
    my $path = shift;
    local *FH; # not my!
    open (FH, $path) or return undef;
    return *FH;
}
$fh = newopen('/etc/passwd');
```

Now that we have the `*foo{THING}` notation, typeglobs aren't used as much for filehandle manipulations, although they're still needed to pass brand new file and directory handles into or out of functions. That's because `*HANDLE{IO}` only works if `HANDLE` has already been used as a handle. In other words, `*FH` must be used to create new symbol table entries; `*foo{THING}` cannot. When in doubt, use `*FH`.

All functions that are capable of creating filehandles (`open()`, `opendir()`, `pipe()`, `socketpair()`, `sysopen()`, `socket()`, and `accept()`) automatically create an anonymous filehandle if the handle passed to them is an uninitialized scalar variable. This allows the constructs such as `open(my $fh, ...)` and `open(local $fh, ...)` to be used to create filehandles that will conveniently be closed automatically when the scope ends, provided there are no other references to them. This largely eliminates the need for typeglobs when opening filehandles that must be passed around, as in the following example:

```
sub myopen {
    open my $fh, "@_"
        or die "Can't open '@_': $!";
    return $fh;
}

{
    my $f = myopen("</etc/motd");
    print <$f>;
    # $f implicitly closed here
}
```

Note that if an initialized scalar variable is used instead the result is different: `my $fh='zzz'; open($fh, ...)` is equivalent to `open(*{'zzz'}, ...)`. use strict 'refs' forbids such practice.

Another way to create anonymous filehandles is with the `Symbol` module or with the `IO::Handle` module and its ilk. These modules have the advantage of not hiding different types of the same name during the `local()`. See the bottom of [open\(\)](#) for an example.

SEE ALSO

See [perlvar](#) for a description of Perl's built-in variables and a discussion of legal variable names. See [perlref](#), [perlsub](#), and [Symbol Tables in perlmod](#) for more discussion on typeglobs and the `*foo{THING}` syntax.

NAME

perldbmfilter – Perl DBM Filters

SYNOPSIS

```
$db = tie %hash, 'DBM', ...

$old_filter = $db->filter_store_key ( sub { ... } ) ;
$old_filter = $db->filter_store_value( sub { ... } ) ;
$old_filter = $db->filter_fetch_key  ( sub { ... } ) ;
$old_filter = $db->filter_fetch_value( sub { ... } ) ;
```

DESCRIPTION

The four `filter_*` methods shown above are available in all the DBM modules that ship with Perl, namely `DB_File`, `GDBM_File`, `NDBM_File`, `ODBM_File` and `SDBM_File`.

Each of the methods work identically, and are used to install (or uninstall) a single DBM Filter. The only difference between them is the place that the filter is installed.

To summarise:

filter_store_key

If a filter has been installed with this method, it will be invoked every time you write a key to a DBM database.

filter_store_value

If a filter has been installed with this method, it will be invoked every time you write a value to a DBM database.

filter_fetch_key

If a filter has been installed with this method, it will be invoked every time you read a key from a DBM database.

filter_fetch_value

If a filter has been installed with this method, it will be invoked every time you read a value from a DBM database.

You can use any combination of the methods from none to all four.

All filter methods return the existing filter, if present, or `undef` in not.

To delete a filter pass `undef` to it.

The Filter

When each filter is called by Perl, a local copy of `$_` will contain the key or value to be filtered. Filtering is achieved by modifying the contents of `$_`. The return code from the filter is ignored.

An Example — the NULL termination problem.

DBM Filters are useful for a class of problems where you *always* want to make the same transformation to all keys, all values or both.

For example, consider the following scenario. You have a DBM database that you need to share with a third-party C application. The C application assumes that *all* keys and values are NULL terminated. Unfortunately when Perl writes to DBM databases it doesn't use NULL termination, so your Perl application will have to manage NULL termination itself. When you write to the database you will have to use something like this:

```
$hash{"$key\0"} = "$value\0" ;
```

Similarly the NULL needs to be taken into account when you are considering the length of existing keys/values.

It would be much better if you could ignore the NULL terminations issue in the main application code and have a mechanism that automatically added the terminating NULL to all keys and values whenever you write to the database and have them removed when you read from the database. As I'm sure you have already guessed, this is a problem that DBM Filters can fix very easily.

```
use strict ;
use warnings ;
use SDBM_File ;
use Fcntl ;

my %hash ;
my $filename = "/tmp/filt" ;
unlink $filename ;

my $db = tie(%hash, 'SDBM_File', $filename, O_RDWR|O_CREAT, 0640)
    or die "Cannot open $filename: $!\n" ;

# Install DBM Filters
$db->filter_fetch_key  ( sub { s/\0$// } ) ;
$db->filter_store_key   ( sub { $_ .= "\0" } ) ;
$db->filter_fetch_value(
    sub { no warnings 'uninitialized' ; s/\0$// } ) ;
$db->filter_store_value( sub { $_ .= "\0" } ) ;

$hash{"abc"} = "def" ;
my $a = $hash{"ABC"} ;
# ...
undef $db ;
untie %hash ;
```

The code above uses SDBM_File, but it will work with any of the DBM modules.

Hopefully the contents of each of the filters should be self-explanatory. Both "fetch" filters remove the terminating NULL, and both "store" filters add a terminating NULL.

Another Example — Key is a C int.

Here is another real-life example. By default, whenever Perl writes to a DBM database it always writes the key and value as strings. So when you use this:

```
$hash{12345} = "something" ;
```

the key 12345 will get stored in the DBM database as the 5 byte string "12345". If you actually want the key to be stored in the DBM database as a C int, you will have to use `pack` when writing, and `unpack` when reading.

Here is a DBM Filter that does it:

```
use strict ;
use warnings ;
use DB_File ;
my %hash ;
my $filename = "/tmp/filt" ;
unlink $filename ;

my $db = tie %hash, 'DB_File', $filename, O_CREAT|O_RDWR, 0666, $DB_HASH
    or die "Cannot open $filename: $!\n" ;

$db->filter_fetch_key  ( sub { $_ = unpack("i", $_) } ) ;
$db->filter_store_key   ( sub { $_ = pack ("i", $_) } ) ;
$hash{123} = "def" ;
# ...
```

```
undef $db ;  
untie %hash ;
```

The code above uses `DB_File`, but again it will work with any of the DBM modules.

This time only two filters have been used — we only need to manipulate the contents of the key, so it wasn't necessary to install any value filters.

SEE ALSO

[DB_File](#), [GDBM_File](#), [NDBM_File](#), [ODBM_File](#) and [SDBM_File](#).

AUTHOR

Paul Marquess

NAME

perldebguts – Guts of Perl debugging

DESCRIPTION

This is not the perldebug(1) manpage, which tells you how to use the debugger. This manpage describes low-level details concerning the debugger's internals, which range from difficult to impossible to understand for anyone who isn't incredibly intimate with Perl's guts. Caveat lector.

Debugger Internals

Perl has special debugging hooks at compile-time and run-time used to create debugging environments. These hooks are not to be confused with the *perl -Dxxx* command described in [perlrun](#), which is usable only if a special Perl is built per the instructions in the *INSTALL* podpage in the Perl source tree.

For example, whenever you call Perl's built-in `caller` function from the package `DB`, the arguments that the corresponding stack frame was called with are copied to the `@DB::args` array. These mechanisms are enabled by calling Perl with the `-d` switch. Specifically, the following additional features are enabled (cf. [\\$^P](#)):

- Perl inserts the contents of `$ENV{PERL5DB}` (or `BEGIN {require 'perl5db.pl'}` if not present) before the first line of your program.
- Each array `@{"_<$filename"}` holds the lines of `$filename` for a file compiled by Perl. The same is also true for `eval`d strings that contain subroutines, or which are currently being executed. The `$filename` for `eval`d strings looks like `(eval 34)`. Code assertions in regexes look like `(re_eval 19)`.

Values in this array are magical in numeric context: they compare equal to zero only if the line is not breakable.

- Each hash `%{"_<$filename"}` contains breakpoints and actions keyed by line number. Individual entries (as opposed to the whole hash) are settable. Perl only cares about Boolean true here, although the values used by *perl5db.pl* have the form `"$break_condition\0$action"`.

The same holds for evaluated strings that contain subroutines, or which are currently being executed. The `$filename` for `eval`d strings looks like `(eval 34)` or `(re_eval 19)`.

- Each scalar `${"_<$filename"}` contains `"_<$filename"`. This is also the case for evaluated strings that contain subroutines, or which are currently being executed. The `$filename` for `eval`d strings looks like `(eval 34)` or `(re_eval 19)`.
- After each required file is compiled, but before it is executed, `DB::postponed(*{"_<$filename"})` is called if the subroutine `DB::postponed` exists. Here, the `$filename` is the expanded name of the required file, as found in the values of `%INC`.
- After each subroutine `subname` is compiled, the existence of `$DB::postponed{subname}` is checked. If this key exists, `DB::postponed(subname)` is called if the `DB::postponed` subroutine also exists.
- A hash `%DB::sub` is maintained, whose keys are subroutine names and whose values have the form `filename:startline-endline`. `filename` has the form `(eval 34)` for subroutines defined inside `eval`s, or `(re_eval 19)` for those within regex code assertions.
- When the execution of your program reaches a point that can hold a breakpoint, the `DB::DB()` subroutine is called if any of the variables `$DB::trace`, `$DB::single`, or `$DB::signal` is true. These variables are not localizable. This feature is disabled when executing inside `DB::DB()`, including functions called from it unless `< $^D & (1<<30)` is true.
- When execution of the program reaches a subroutine call, a call to `&DB::sub(args)` is made instead, with `$DB::sub` holding the name of the called subroutine. (This doesn't happen if the subroutine was compiled in the `DB` package.)

Note that if `&DB:::sub` needs external data for it to work, no subroutine call is possible without it. As an example, the standard debugger's `&DB:::sub` depends on the `$DB:::deep` variable (it defines how many levels of recursion deep into the debugger you can go before a mandatory break). If `$DB:::deep` is not defined, subroutine calls are not possible, even though `&DB:::sub` exists.

Writing Your Own Debugger

=head3 Environment Variables

The `PERL5DB` environment variable can be used to define a debugger. For example, the minimal "working" debugger (it actually doesn't do anything) consists of one line:

```
sub DB::DB {}
```

It can easily be defined like this:

```
$ PERL5DB="sub DB::DB {}" perl -d your-script
```

Another brief debugger, slightly more useful, can be created with only the line:

```
sub DB::DB {print ++$i; scalar <STDIN>}
```

This debugger prints a number which increments for each statement encountered and waits for you to hit a newline before continuing to the next statement.

The following debugger is actually useful:

```
{
  package DB;
  sub DB {}
  sub sub {print ++$i, " $sub\n"; &$sub}
}
```

It prints the sequence number of each subroutine call and the name of the called subroutine. Note that `&DB:::sub` is being compiled into the package `DB` through the use of the `package` directive.

When it starts, the debugger reads your rc file (`./perldb` or `~/perldb` under Unix), which can set important options. (A subroutine (`&afterinit`) can be defined here as well; it is executed after the debugger completes its own initialization.)

After the rc file is read, the debugger reads the `PERLDB_OPTS` environment variable and uses it to set debugger options. The contents of this variable are treated as if they were the argument of an `o ...` debugger command (q.v. in [Options](#)).

=head3 Debugger internal variables In addition to the file and subroutine-related variables mentioned above, the debugger also maintains various magical internal variables.

- `@DB::dbline` is an alias for `@{":::<current_file">}`, which holds the lines of the currently-selected file (compiled by Perl), either explicitly chosen with the debugger's `f` command, or implicitly by flow of execution.

Values in this array are magical in numeric context: they compare equal to zero only if the line is not breakable.

- `%DB::dbline`, is an alias for `%{":::<current_file">}`, which contains breakpoints and actions keyed by line number in the currently-selected file, either explicitly chosen with the debugger's `f` command, or implicitly by flow of execution.

As previously noted, individual entries (as opposed to the whole hash) are settable. Perl only cares about Boolean true here, although the values used by *perl5db.pl* have the form `"$break_condition\0$action"`.

=head3 Debugger customization functions

Some functions are provided to simplify customization.

- See [Options in perldebug](#) for description of options parsed by `DB::parse_options(string)` parses debugger options; see [Options](#) for a description of options recognized.
- `DB::dump_trace(skip[,count])` skips the specified number of frames and returns a list containing information about the calling frames (all of them, if `count` is missing). Each entry is reference to a hash with keys `context` (either `..`, `$`, or `@`), `sub` (subroutine name, or info about `eval`), `args` (undef or a reference to an array), `file`, and `line`.
- `DB::print_trace(FH, skip[, count[, short]])` prints formatted info about caller frames. The last two functions may be convenient as arguments to `< <`, `< <<` commands.

Note that any variables and functions that are not documented in this manpages (or in [perldebug](#)) are considered for internal use only, and as such are subject to change without notice.

Frame Listing Output Examples

The frame option can be used to control the output of frame information. For example, contrast this expression trace:

```
$ perl -de 42
Stack dump during die enabled outside of evals.

Loading DB routines from perl5db.pl patch level 0.94
Emacs support available.

Enter h or 'h h' for help.

main::(-e:1): 0
  DB<1> sub foo { 14 }

  DB<2> sub bar { 3 }

  DB<3> t print foo() * bar()
main::((eval 172):3): print foo() + bar();
main::foo((eval 168):2):
main::bar((eval 170):2):
42
```

with this one, once the option `frame=2` has been set:

```
DB<4> o f=2
          frame = '2'
DB<5> t print foo() * bar()
3:      foo() * bar()
entering main::foo
  2:      sub foo { 14 };
exited main::foo
entering main::bar
  2:      sub bar { 3 };
exited main::bar
42
```

By way of demonstration, we present below a laborious listing resulting from setting your `PERLDB_OPTS` environment variable to the value `f=n N`, and running `perl -d -V` from the command line. Examples use various values of `n` are shown to give you a feel for the difference between settings. Long those it may be, this is not a complete listing, but only excerpts.

```
1
    entering main::BEGIN
    entering Config::BEGIN
    Package lib/Exporter.pm.
    Package lib/Carp.pm.
```

```

Package lib/Config.pm.
entering Config::TIEHASH
entering Exporter::import
  entering Exporter::export
entering Config::myconfig
entering Config::FETCH
entering Config::FETCH
entering Config::FETCH
entering Config::FETCH

```

2

```

entering main::BEGIN
  entering Config::BEGIN
    Package lib/Exporter.pm.
    Package lib/Carp.pm.
  exited Config::BEGIN
  Package lib/Config.pm.
  entering Config::TIEHASH
  exited Config::TIEHASH
  entering Exporter::import
    entering Exporter::export
    exited Exporter::export
  exited Exporter::import
exited main::BEGIN
entering Config::myconfig
  entering Config::FETCH
  exited Config::FETCH
  entering Config::FETCH
  exited Config::FETCH
  entering Config::FETCH

```

4

```

in $=main::BEGIN() from /dev/null:0
in $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
  Package lib/Config.pm.
in $=Config::TIEHASH('Config') from lib/Config.pm:644
in $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
  in $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from li
in @=Config::myconfig() from /dev/null:0
  in $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
  in $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
  in $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
  in $=Config::FETCH(ref(Config), 'PERL_SUBVERSION') from lib/Config.pm:574
  in $=Config::FETCH(ref(Config), 'osname') from lib/Config.pm:574
  in $=Config::FETCH(ref(Config), 'osvers') from lib/Config.pm:574

```

6

```

in $=main::BEGIN() from /dev/null:0
in $=Config::BEGIN() from lib/Config.pm:2
  Package lib/Exporter.pm.
  Package lib/Carp.pm.
out $=Config::BEGIN() from lib/Config.pm:0
Package lib/Config.pm.

```

```

in  $=Config::TIEHASH('Config') from lib/Config.pm:644
out $=Config::TIEHASH('Config') from lib/Config.pm:644
in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
out $=main::BEGIN() from /dev/null:0
in  @=Config::myconfig() from /dev/null:0
in  $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
out $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
in  $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
out $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
in  $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
out $=Config::FETCH(ref(Config), 'PERL_VERSION') from lib/Config.pm:574
in  $=Config::FETCH(ref(Config), 'PERL_SUBVERSION') from lib/Config.pm:574

```

14

```

in  $=main::BEGIN() from /dev/null:0
in  $=Config::BEGIN() from lib/Config.pm:2
    Package lib/Exporter.pm.
    Package lib/Carp.pm.
out $=Config::BEGIN() from lib/Config.pm:0
    Package lib/Config.pm.
in  $=Config::TIEHASH('Config') from lib/Config.pm:644
out $=Config::TIEHASH('Config') from lib/Config.pm:644
in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
out $=main::BEGIN() from /dev/null:0
in  @=Config::myconfig() from /dev/null:0
in  $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Config.pm:574
out $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from lib/Config.pm:574
in  $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Config.pm:574
out $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from lib/Config.pm:574

```

30

```

in  $=CODE(0x15eca4)() from /dev/null:0
in  $=CODE(0x182528)() from lib/Config.pm:2
    Package lib/Exporter.pm.
out $=CODE(0x182528)() from lib/Config.pm:0
    scalar context return from CODE(0x182528): undef
    Package lib/Config.pm.
in  $=Config::TIEHASH('Config') from lib/Config.pm:628
out $=Config::TIEHASH('Config') from lib/Config.pm:628
    scalar context return from Config::TIEHASH: empty hash
in  $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
in  $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
out $=Exporter::export('Config', 'main', 'myconfig', 'config_vars') from lib/
    scalar context return from Exporter::export: ''
out $=Exporter::import('Config', 'myconfig', 'config_vars') from /dev/null:0
    scalar context return from Exporter::import: ''

```

In all cases shown above, the line indentation shows the call tree. If bit 2 of frame is set, a line is printed on exit from a subroutine as well. If bit 4 is set, the arguments are printed along with the caller info. If bit 8 is set, the arguments are printed even if they are tied or references. If bit 16 is set, the return value is printed,

too.

When a package is compiled, a line like this

```
Package lib/Carp.pm.
```

is printed with proper indentation.

Debugging regular expressions

There are two ways to enable debugging output for regular expressions.

If your perl is compiled with `-DDEBUGGING`, you may use the `-Dr` flag on the command line.

Otherwise, one can use `re 'debug'`, which has effects at compile time and run time. It is not lexically scoped.

Compile-time output

The debugging output at compile time looks like this:

```
Compiling REx '[bc]d(ef*g)+h[ij]k$'
size 45 Got 364 bytes for offset annotations.
first at 1
rarest char g at 0
rarest char d at 0
  1: ANYOF[bc] (12)
 12: EXACT <d> (14)
 14: CURLYX[0] {1,32767} (28)
 16:   OPEN1 (18)
 18:     EXACT <e> (20)
 20:     STAR (23)
 21:       EXACT <f> (0)
 23:       EXACT <g> (25)
 25:   CLOSE1 (27)
 27:   WHILEM[1/1] (0)
 28: NOTHING (29)
 29: EXACT <h> (31)
 31: ANYOF[ij] (42)
 42: EXACT <k> (44)
 44: EOL (45)
 45: END (0)
anchored 'de' at 1 floating 'gh' at 3..2147483647 (checking floating)
  stclass 'ANYOF[bc]' minlen 7
Offsets: [45]
  1[4] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 5[1]
  0[0] 12[1] 0[0] 6[1] 0[0] 7[1] 0[0] 9[1] 8[1] 0[0] 10[1] 0[0]
 11[1] 0[0] 12[0] 12[0] 13[1] 0[0] 14[4] 0[0] 0[0] 0[0] 0[0]
  0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 18[1] 0[0] 19[1] 20[0]
Omitting '$' '$&' '$' support.
```

The first line shows the pre-compiled form of the regex. The second shows the size of the compiled form (in arbitrary units, usually 4-byte words) and the total number of bytes allocated for the offset/length table, usually $4 + \text{size} * 8$. The next line shows the label *id* of the first node that does a match.

The

```
anchored 'de' at 1 floating 'gh' at 3..2147483647 (checking floating)
  stclass 'ANYOF[bc]' minlen 7
```

line (split into two lines above) contains optimizer information. In the example shown, the optimizer found that the match should contain a substring `de` at offset 1, plus substring `gh` at some offset between 3 and

infinity. Moreover, when checking for these substrings (to abandon impossible matches quickly), Perl will check for the substring `gh` before checking for the substring `de`. The optimizer may also use the knowledge that the match starts (at the *first id*) with a character class, and no string shorter than 7 characters can possibly match.

The fields of interest which may appear in this line are

anchored *STRING* at *POS*

floating *STRING* at *POS1..POS2*

See above.

matching floating/anchored

Which substring to check first.

minlen

The minimal length of the match.

stclass *TYPE*

Type of first matching node.

noscan

Don't scan for the found substrings.

isall

Means that the optimizer information is all that the regular expression contains, and thus one does not need to enter the regex engine at all.

GPOS

Set if the pattern contains `\G`.

plus

Set if the pattern starts with a repeated char (as in `x+y`).

implicit

Set if the pattern starts with `.*`.

with eval

Set if the pattern contain eval-groups, such as `(?{ code })` and `(??{ code })`.

anchored(*TYPE*)

If the pattern may match only at a handful of places, (with *TYPE* being `BOL`, `MBOL`, or `GPOS`. See the table below.

If a substring is known to match at end-of-line only, it may be followed by `$`, as in floating ``k`$`.

The optimizer-specific information is used to avoid entering (a slow) regex engine on strings that will not definitely match. If the `isall` flag is set, a call to the regex engine may be avoided even when the optimizer found an appropriate place for the match.

Above the optimizer section is the list of *nodes* of the compiled form of the regex. Each line has format

id: *TYPE* *OPTIONAL-INFO* (*next-id*)

Types of nodes

Here are the possible types, with short descriptions:

```
# TYPE arg-description [num-args] [longjump-len] DESCRIPTION
# Exit points
END          no      End of program.
SUCCEED      no      Return from a subroutine, basically.
```

```

# Anchors:
BOL      no      Match "" at beginning of line.
MBOLno   Same, assuming multiline.
SBOLno   Same, assuming singleline.
EOS      no      Match "" at end of string.
EOL      no      Match "" at end of line.
MEOLno   Same, assuming multiline.
SEOLno   Same, assuming singleline.
BOUNDno  Match "" at any word boundary
BOUNDL   no      Match "" at any word boundary
NBOUND   no      Match "" at any word non-boundary
NBOUNDL  no      Match "" at any word non-boundary
GPOSno   Matches where last m//g left off.

# [Special] alternatives
ANY      no      Match any one character (except newline).
SANY     no      Match any one character.
ANYOF    sv      Match character in (or not in) this class.
ALNUM    no      Match any alphanumeric character
ALNUML   no      Match any alphanumeric char in locale
NALNUM   no      Match any non-alphanumeric character
NALNUML  no      Match any non-alphanumeric char in locale
SPACE    no      Match any whitespace character
SPACEL   no      Match any whitespace char in locale
NSPACE   no      Match any non-whitespace character
NSPACEL  no      Match any non-whitespace char in locale
DIGIT    no      Match any numeric character
NDIGIT   no      Match any non-numeric character

# BRANCH    The set of branches constituting a single choice are hooked
#            together with their "next" pointers, since precedence prevents
#            anything being concatenated to any individual branch.  The
#            "next" pointer of the last BRANCH in a choice points to the
#            thing following the whole choice.  This is also where the
#            final "next" pointer of each individual branch points; each
#            branch starts with the operand node of a BRANCH node.
#
BRANCH    node    Match this alternative, or the next...

# BACK      Normal "next" pointers all implicitly point forward; BACK
#            exists to make loop structures possible.
# not used
BACK      no      Match "", "next" ptr points backward.

# Literals
EXACT     sv      Match this string (preceded by length).
EXACTF    sv      Match this string, folded (prec. by length).
EXACTFL   sv      Match this string, folded in locale (w/len).

# Do nothing
NOTHING   no      Match empty string.
# A variant of above which delimits a group, thus stops optimizations
TAIL      no      Match empty string. Can jump here from outside.

# STAR, PLUS '?' , and complex '*' and '+', are implemented as circular
#            BRANCH structures using BACK.  Simple cases (one character
#            per match) are implemented with STAR and PLUS for speed

```

```

#           and to minimize recursive plunges.
#
STARnode   Match this (simple) thing 0 or more times.
PLUSnode   Match this (simple) thing 1 or more times.

CURLY      sv 2    Match this simple thing {n,m} times.
CURLYN     no 2    Match next-after-this simple thing
#           {n,m} times, set parens.
CURLYM     no 2    Match this medium-complex thing {n,m} times.
CURLYX     sv 2    Match this complex thing {n,m} times.

# This terminator creates a loop structure for CURLYX
WHILEM     no      Do curly processing and see if rest matches.

# OPEN,CLOSE,GROUPP ...are numbered at compile time.
OPEN       num 1   Mark this point in input as start of #n.
CLOSE      num 1   Analogous to OPEN.

REF        num 1   Match some already matched string
REFF       num 1   Match already matched string, folded
REFFL      num 1   Match already matched string, folded in loc.

# grouping assertions
IFMATCH    off 1 2 Succeeds if the following matches.
UNLESSM    off 1 2 Fails if the following matches.
SUSPEND    off 1 1 "Independent" sub-regex.
IFTHEN     off 1 1 Switch, should be preceded by switcher .
GROUPP     num 1   Whether the group matched.

# Support for long regex
LONGJMP     off 1 1 Jump far away.
BRANCHJ     off 1 1 BRANCH with long offset.

# The heavy worker
EVAL       evl 1   Execute some Perl code.

# Modifiers
MINMOD      no      Next operator is not greedy.
LOGICAL     no      Next opcode should set the flag only.

# This is not used yet
RENUM      off 1 1 Group with independently numbered parens.

# This is not really a node, but an optimized away piece of a "long" node.
# To simplify debugging output, we mark it as if it were a node
OPTIMIZED  off      Placeholder for dump.

```

=for unprinted-credits Next section M-J. Dominus (mjd-perl-patch+@plover.com) 20010421

Following the optimizer information is a dump of the offset/length table, here split across several lines:

```

Offsets: [45]
  1[4] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 5[1]
  0[0] 12[1] 0[0] 6[1] 0[0] 7[1] 0[0] 9[1] 8[1] 0[0] 10[1] 0[0]
  11[1] 0[0] 12[0] 12[0] 13[1] 0[0] 14[4] 0[0] 0[0] 0[0] 0[0]
  0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 18[1] 0[0] 19[1] 20[0]

```

The first line here indicates that the offset/length table contains 45 entries. Each entry is a pair of integers, denoted by offset [length]. Entries are numbered starting with 1, so entry #1 here is 1[4] and entry #12 is 5[1]. 1[4] indicates that the node labeled 1: (the 1: ANYOF[bc]) begins at character position 1 in the pre-compiled form of the regex, and has a length of 4 characters. 5[1] in position 12 indicates that

the node labeled 12: (the < 12: EXACT <d>) begins at character position 5 in the pre-compiled form of the regex, and has a length of 1 character. 12[1] in position 14 indicates that the node labeled 14: (the < 14: CURLYX[0] {1,32767}) begins at character position 12 in the pre-compiled form of the regex, and has a length of 1 character—that is, it corresponds to the + symbol in the precompiled regex.

0[0] items indicate that there is no corresponding node.

Run-time output

First of all, when doing a match, one may get no run-time output even if debugging is enabled. This means that the regex engine was never entered and that all of the job was therefore done by the optimizer.

If the regex engine was entered, the output may look like this:

```
Matching `[bc]d(ef*g)+h[ij]k$` against `abcdefg__gh__`
Setting an EVAL scope, savestack=3
  2 <ab> <cdefg__gh__> | 1: ANYOF
  3 <abc> <defg__gh__> | 11: EXACT <d>
  4 <abcd> <efg__gh__> | 13: CURLYX {1,32767}
  4 <abcd> <efg__gh__> | 26: WHILEM
                        0 out of 1..32767 cc=ffffff31c
  4 <abcd> <efg__gh__> | 15: OPEN1
  4 <abcd> <efg__gh__> | 17: EXACT <e>
  5 <abcde> <fg__gh__> | 19: STAR
                        EXACT <f> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
  6 <bcdef> <g__gh__> | 22: EXACT <g>
  7 <bcdefg> <__gh__> | 24: CLOSE1
  7 <bcdefg> <__gh__> | 26: WHILEM
                        1 out of 1..32767 cc=ffffff31c
Setting an EVAL scope, savestack=12
  7 <bcdefg> <__gh__> | 15: OPEN1
  7 <bcdefg> <__gh__> | 17: EXACT <e>
    restoring \1 to 4(4)..7
                                failed, try continuation...
  7 <bcdefg> <__gh__> | 27: NOTHING
  7 <bcdefg> <__gh__> | 28: EXACT <h>
                                failed...
                                failed...
```

The most significant information in the output is about the particular *node* of the compiled regex that is currently being tested against the target string. The format of these lines is

STRING-OFFSET <*PRE-STRING* <*POST-STRING* |*ID*: *TYPE*

The *TYPE* info is indented with respect to the backtracking level. Other incidental information appears interspersed within.

Debugging Perl memory usage

Perl is a profligate wastrel when it comes to memory use. There is a saying that to estimate memory usage of Perl, assume a reasonable algorithm for memory allocation, multiply that estimate by 10, and while you still may miss the mark, at least you won't be quite so astonished. This is not absolutely true, but may provide a good grasp of what happens.

Assume that an integer cannot take less than 20 bytes of memory, a float cannot take less than 24 bytes, a string cannot take less than 32 bytes (all these examples assume 32-bit architectures, the result are quite a bit worse on 64-bit architectures). If a variable is accessed in two of three different ways (which require an integer, a float, or a string), the memory footprint may increase yet another 20 bytes. A sloppy malloc(3) implementation can inflate these numbers dramatically.

On the opposite end of the scale, a declaration like

```
sub foo;
```

may take up to 500 bytes of memory, depending on which release of Perl you're running.

Anecdotal estimates of source-to-compiled code bloat suggest an eightfold increase. This means that the compiled form of reasonable (normally commented, properly indented etc.) code will take about eight times more space in memory than the code took on disk.

There are two Perl-specific ways to analyze memory usage: `$ENV{PERL_DEBUG_MSTATS}` and `-DL` command-line switch. The first is available only if Perl is compiled with Perl's `malloc()`; the second only if Perl was built with `-DDEBUGGING`. See the instructions for how to do this in the *INSTALL* podpage at the top level of the Perl source tree.

Using `$ENV{PERL_DEBUG_MSTATS}`

If your perl is using Perl's `malloc()` and was compiled with the necessary switches (this is the default), then it will print memory usage statistics after compiling your code when `<`

`$ENV{PERL_DEBUG_MSTATS} 1`, and before termination of the program when `<`

`$ENV{PERL_DEBUG_MSTATS} = 1`. The report format is similar to the following example:

```
$ PERL_DEBUG_MSTATS=2 perl -e "require Carp"
Memory allocation statistics after compilation: (buckets 4(4)..8188(8192)
 14216 free:   130   117   28    7    9    0    2    2    1 0 0
              437    61    36    0    5
 60924 used:   125   137   161   55    7    8    6   16    2 0 1
              74   109   304    84   20
Total sbrk(): 77824/21:119. Odd ends: pad+heads+chain+tail: 0+636+0+2048.
Memory allocation statistics after execution:   (buckets 4(4)..8188(8192)
 30888 free:   245    78    85   13    6    2    1    3    2 0 1
              315   162    39   42   11
175816 used:   265   176  1112  111   26   22   11   27    2 1 1
              196   178  1066   798   39
Total sbrk(): 215040/47:145. Odd ends: pad+heads+chain+tail: 0+2192+0+6144.
```

It is possible to ask for such a statistic at arbitrary points in your execution using the `mstat()` function out of the standard `Devel::Peek` module.

Here is some explanation of that format:

`buckets SMALLEST (APPROX) .. GREATEST (APPROX)`

Perl's `malloc()` uses bucketed allocations. Every request is rounded up to the closest bucket size available, and a bucket is taken from the pool of buckets of that size.

The line above describes the limits of buckets currently in use. Each bucket has two sizes: memory footprint and the maximal size of user data that can fit into this bucket. Suppose in the above example that the smallest bucket were size 4. The biggest bucket would have usable size 8188, and the memory footprint would be 8192.

In a Perl built for debugging, some buckets may have negative usable size. This means that these buckets cannot (and will not) be used. For larger buckets, the memory footprint may be one page greater than a power of 2. If so, case the corresponding power of two is printed in the `APPROX` field above.

Free/Used

The 1 or 2 rows of numbers following that correspond to the number of buckets of each size between `SMALLEST` and `GREATEST`. In the first row, the sizes (memory footprints) of buckets are powers of two—or possibly one page greater. In the second row, if present, the memory footprints of the buckets are between the memory footprints of two buckets "above".

For example, suppose under the previous example, the memory footprints were

```

free:      8      16      32      64      128    256 512 1024 2048 4096 8192
          4      12      24      48      80

```

With non-DEBUGGING perl, the buckets starting from 128 have a 4-byte overhead, and thus an 8192-long bucket may take up to 8188-byte allocations.

Total sbrk(): SBRKed/SBRKs:CONTINUOUS

The first two fields give the total amount of memory perl sbrk(2)ed (ess-broken? :-) and number of sbrk(2)s used. The third number is what perl thinks about continuity of returned chunks. So long as this number is positive, malloc() will assume that it is probable that sbrk(2) will provide continuous memory.

Memory allocated by external libraries is not counted.

pad: 0

The amount of sbrk(2)ed memory needed to keep buckets aligned.

heads: 2192

Although memory overhead of bigger buckets is kept inside the bucket, for smaller buckets, it is kept in separate areas. This field gives the total size of these areas.

chain: 0

malloc() may want to subdivide a bigger bucket into smaller buckets. If only a part of the deceased bucket is left unsubdivided, the rest is kept as an element of a linked list. This field gives the total size of these chunks.

tail: 6144

To minimize the number of sbrk(2)s, malloc() asks for more memory. This field gives the size of the yet unused part, which is sbrk(2)ed, but never touched.

Example of using -DL switch

Below we show how to analyse memory usage by

```
do 'lib/auto/POSIX/autosplit.ix';
```

The file in question contains a header and 146 lines similar to

```
sub getcwd;
```

WARNING: The discussion below supposes 32-bit architecture. In newer releases of Perl, memory usage of the constructs discussed here is greatly improved, but the story discussed below is a real-life story. This story is mercilessly terse, and assumes rather more than cursory knowledge of Perl internals. Type space to continue, 'q' to quit. (Actually, you just want to skip to the next section.)

Here is the itemized list of Perl allocations performed during parsing of this file:

```

!!! "after" at test.pl line 3.
  Id  subtot   4   8  12  16  20  24  28  32  36  40  48  56  64  72  80  80+
0 02  13752   .   .   .   . 294   .   .   .   .   .   .   .   .   .   .   4
0 54   5545   .   .   8 124  16   .   .   .   1   1   .   .   .   .   .   3
5 05     32   .   .   .   .   .   .   .   1   .   .   .   .   .   .   .
6 02   7152   .   .   .   .   .   .   .   .   .   . 149   .   .   .   .
7 02   3600   .   .   .   .   . 150   .   .   .   .   .   .   .   .   .
7 03     64   .  -1   .   1   .   .   2   .   .   .   .   .   .   .
7 04   7056   .   .   .   .   .   .   .   .   .   .   .   .   .   .   7
7 17  38404   .   .   .   .   .   .   .   1   .   . 442 149   .   . 147   .
9 03   2078  17 249  32   .   .   .   .   2   .   .   .   .   .   .   .

```

To see this list, insert two warn('!...') statements around the call:

```
warn('!');
do 'lib/auto/POSIX/autosplit.ix';
warn('!!! "after"');
```

and run it with Perl's **-DL** option. The first `warn()` will print memory allocation info before parsing the file and will memorize the statistics at this point (we ignore what it prints). The second `warn()` prints increments with respect to these memorized data. This is the printout shown above.

Different *Ids* on the left correspond to different subsystems of the perl interpreter. They are just the first argument given to the perl memory allocation API named `New()`. To find what 9 03 means, just **grep** the perl source for 903. You'll find it in *util.c*, function `savepv()`. (I know, you wonder why we told you to **grep** and then gave away the answer. That's because grepping the source is good for the soul.) This function is used to store a copy of an existing chunk of memory. Using a C debugger, one can see that the function was called either directly from `gv_init()` or via `sv_magic()`, and that `gv_init()` is called from `gv_fetchpv()`—which was itself called from `newSUB()`. Please stop to catch your breath now.

NOTE: To reach this point in the debugger and skip the calls to `savepv()` during the compilation of the main program, you should set a C breakpoint in `Perl_warn()`, continue until this point is reached, and *then* set a C breakpoint in `Perl_savepv()`. Note that you may need to skip a handful of `Perl_savepv()` calls that do not correspond to mass production of CVs (there are more 903 allocations than 146 similar lines of *lib/auto/POSIX/autosplit.ix*). Note also that `Perl_` prefixes are added by macroization code in perl header files to avoid conflicts with external libraries.

Anyway, we see that 903 ids correspond to creation of globs, twice per glob – for glob name, and glob stringification magic.

Here are explanations for other *Ids* above:

717

Creates bigger XPV* structures. In the case above, it creates 3 AVs per subroutine, one for a list of lexical variable names, one for a scratchpad (which contains lexical variables and targets), and one for the array of scratchpads needed for recursion.

It also creates a GV and a CV per subroutine, all called from `start_subparse()`.

002 Creates a C array corresponding to the AV of scratchpads and the scratchpad itself. The first fake entry of this scratchpad is created though the subroutine itself is not defined yet.

It also creates C arrays to keep data for the stash. This is one HV, but it grows; thus, there are 4 big allocations: the big chunks are not freed, but are kept as additional arenas for SV allocations.

054 Creates a HEK for the name of the glob for the subroutine. This name is a key in a *stash*.

Big allocations with this *Id* correspond to allocations of new arenas to keep HE.

602 Creates a GP for the glob for the subroutine.

702 Creates the MAGIC for the glob for the subroutine.

704 Creates *arenas* which keep SVs.

-DL details

If Perl is run with **-DL** option, then `warn()`s that start with '!' behave specially. They print a list of *categories* of memory allocations, and statistics of allocations of different sizes for these categories.

If `warn()` string starts with

!!!

print changed categories only, print the differences in counts of allocations.

!! print grown categories only; print the absolute values of counts, and totals.

! print nonempty categories, print the absolute values of counts and totals.

Limitations of `-DL` statistics

If an extension or external library does not use the Perl API to allocate memory, such allocations are not counted.

SEE ALSO

perldebug, *perlguts*, *perlrun re*, and *Devel::DProf*.

NAME

perldebtut – Perl debugging tutorial

DESCRIPTION

A (very) lightweight introduction in the use of the perl debugger, and a pointer to existing, deeper sources of information on the subject of debugging perl programs.

There's an extraordinary number of people out there who don't appear to know anything about using the perl debugger, though they use the language every day. This is for them.

use strict

First of all, there's a few things you can do to make your life a lot more straightforward when it comes to debugging perl programs, without using the debugger at all. To demonstrate, here's a simple script, named "hello", with a problem:

```
#!/usr/bin/perl

$var1 = 'Hello World'; # always wanted to do that :-)
$var2 = "$var1\n";

print $var2;
exit;
```

While this compiles and runs happily, it probably won't do what's expected, namely it doesn't print "Hello World\n" at all; It will on the other hand do exactly what it was told to do, computers being a bit that way inclined. That is, it will print out a newline character, and you'll get what looks like a blank line. It looks like there's 2 variables when (because of the typo) there's really 3:

```
$var1 = 'Hello World';
$var1 = undef;
$var2 = "\n";
```

To catch this kind of problem, we can force each variable to be declared before use by pulling in the strict module, by putting 'use strict;' after the first line of the script.

Now when you run it, perl complains about the 3 undeclared variables and we get four error messages because one variable is referenced twice:

```
Global symbol "$var1" requires explicit package name at ./t1 line 4.
Global symbol "$var2" requires explicit package name at ./t1 line 5.
Global symbol "$var1" requires explicit package name at ./t1 line 5.
Global symbol "$var2" requires explicit package name at ./t1 line 7.
Execution of ./hello aborted due to compilation errors.
```

Luvverly! and to fix this we declare all variables explicitly and now our script looks like this:

```
#!/usr/bin/perl
use strict;

my $var1 = 'Hello World';
my $var1 = undef;
my $var2 = "$var1\n";

print $var2;
exit;
```

We then do (always a good idea) a syntax check before we try to run it again:

```
> perl -c hello
hello syntax OK
```

And now when we run it, we get "\n" still, but at least we know why. Just getting this script to compile has

exposed the '\$var1' (with the letter 'l') variable, and simply changing \$varl to \$var1 solves the problem.

Looking at data and -w and v

Ok, but how about when you want to really see your data, what's in that dynamic variable, just before using it?

```
#!/usr/bin/perl
use strict;

my $key = 'welcome';
my %data = (
    'this' => qw(that),
    'tom' => qw(and jerry),
    'welcome' => q(Hello World),
    'zip' => q(welcome),
);
my @data = keys %data;

print "$data{$key}\n";
exit;
```

Looks OK, after it's been through the syntax check (perl -c scriptname), we run it and all we get is a blank line again! Hmmmm.

One common debugging approach here, would be to liberally sprinkle a few print statements, to add a check just before we print out our data, and another just after:

```
print "All OK\n" if grep($key, keys %data);
print "$data{$key}\n";
print "done: '$data{$key}'\n";
```

And try again:

```
> perl data
All OK

done: ''
```

After much staring at the same piece of code and not seeing the wood for the trees for some time, we get a cup of coffee and try another approach. That is, we bring in the cavalry by giving perl the '-d' switch on the command line:

```
> perl -d data
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main:.(./data:4):      my $key = 'welcome';
```

Now, what we've done here is to launch the built-in perl debugger on our script. It's stopped at the first line of executable code and is waiting for input.

Before we go any further, you'll want to know how to quit the debugger: use just the letter 'q', not the words 'quit' or 'exit':

```
DB<1> q
>
```

That's it, you're back on home turf again.

help

Fire the debugger up again on your script and we'll look at the help menu. There's a couple of ways of calling help: a simple **'h'** will get the summary help list, **'lh'** (pipe-h) will pipe the help through your pager (which is (probably 'more' or 'less'), and finally, **'h h'** (h-space-h) will give you the entire help screen. Here is the summary page:

D1h

List/search source lines:	Control script execution:
l [ln sub] List source code	T Stack trace
- or . List previous/current line	s [expr] Single step [in expr]
v [line] View around line	n [expr] Next, steps over subs
f filename View source in file	<CR/Enter> Repeat last n or s
/pattern/ ?patt? Search forw/backw	r Return from subroutine
M Show module versions	c [ln sub] Continue until position
Debugger controls:	L List break/watch/actions
o [...] Set debugger options	t [expr] Toggle trace [trace expr]
<[<] {[{}] >[>] [cmd] Do pre/post-prompt	b [ln event sub] [cnd] Set breakpoint
! [N pat] Redo a previous command	B ln * Delete a/all breakpoints
H [-num] Display last num commands	a [ln] cmd Do cmd before line
= [a val] Define/list an alias	A ln * Delete a/all actions
h [db_cmd] Get help on command	w expr Add a watch expression
h h Complete help page	W expr * Delete a/all watch exprs
[]db_cmd Send output to pager	![!] syscmd Run cmd in a subprocess
q or ^D Quit	R Attempt a restart
Data Examination:	expr Execute perl code, also see: s,n,t expr
x m expr Evals expr in list context, dumps the result or lists methods.	
p expr Print expression (uses script's current package).	
S [![]pat] List subroutine names [not] matching pattern	
V [Pk [Vars]] List Variables in Package. Vars can be ~pattern or !pattern.	
X [Vars] Same as "V current_package [Vars]".	
y [n [Vars]] List lexicals in higher scope <n>. Vars same as V.	

For more help, type h cmd_letter, or run man perldebug for all docs.

More confusing options than you can shake a big stick at! It's not as bad as it looks and it's very useful to know more about all of it, and fun too!

There's a couple of useful ones to know about straight away. You wouldn't think we're using any libraries at all at the moment, but **'M'** will show which modules are currently loaded, and their version number, while **'m'** will show the methods, and **'S'** shows all subroutines (by pattern) as shown below. **'V'** and **'X'** show variables in the program by package scope and can be constrained by pattern.

```
DB<2>S str
dumpvar::stringify
strict::bits
strict::import
strict::unimport
```

Using **'X'** and cousins requires you not to use the type identifiers (\$@%) , just the 'name':

```
DM<3>X ~err
FileHandle(stderr) => fileno(2)
```

Remember we're in our tiny program with a problem, we should have a look at where we are, and what our data looks like. First of all let's view some code at our present position (the first line of code in this case), via **'v'**:

```
DB<4> v
```



```

1      #!/usr/bin/perl
2:      use strict;
3
4==>    my $key = 'welcome';
5:      my %data = (
6          'this' => qw(that),
7          'tom'  => qw(and jerry),
8          'welcome' => q(Hello World),
9          'zip'  => q(welcome),
10         );

```

At line number 4 is a helpful pointer, that tells you where you are now. To see more code, type 'v' again:

```

DB<4> v
8          'welcome' => q(Hello World),
9          'zip'  => q(welcome),
10         );
11:    my @data = keys %data;
12:    print "All OK\n" if grep($key, keys %data);
13:    print "$data{$key}\n";
14:    print "done: '$data{$key}'\n";
15:    exit;

```

And if you wanted to list line 5 again, type 'l 5', (note the space):

```

DB<4> l 5
5:      my %data = (

```

In this case, there's not much to see, but of course normally there's pages of stuff to wade through, and 'l' can be very useful. To reset your view to the line we're about to execute, type a lone period '.':

```

DB<5> .
main::(/data_a:4):    my $key = 'welcome';

```

The line shown is the one that is about to be executed **next**, it hasn't happened yet. So while we can print a variable with the letter 'p', at this point all we'd get is an empty (undefined) value back. What we need to do is to step through the next executable statement with an 's':

```

DB<6> s
main::(/data_a:5):    my %data = (
main::(/data_a:6):    'this' => qw(that),
main::(/data_a:7):    'tom'  => qw(and jerry),
main::(/data_a:8):    'welcome' => q(Hello World),
main::(/data_a:9):    'zip'  => q(welcome),
main::(/data_a:10):   );

```

Now we can have a look at that first (\$key) variable:

```

DB<7> p $key
welcome

```

line 13 is where the action is, so let's continue down to there via the letter 'c', which by the way, inserts a 'one-time-only' breakpoint at the given line or sub routine:

```

DB<8> c 13
All OK
main::(/data_a:13):    print "$data{$key}\n";

```

We've gone past our check (where 'All OK' was printed) and have stopped just before the meat of our task. We could try to print out a couple of variables to see what is happening:

```
DB<9> p $data{$key}
```

Not much in there, lets have a look at our hash:

```
DB<10> p %data
Hello Worldziptomandwelcomejerrywelcomethisthat
```

```
DB<11> p keys %data
Hello Worldtomwelcomejerrythis
```

Well, this isn't very easy to read, and using the helpful manual (**h h**), the '**x**' command looks promising:

```
DB<12> x %data
0  'Hello World'
1  'zip'
2  'tom'
3  'and'
4  'welcome'
5  undef
6  'jerry'
7  'welcome'
8  'this'
9  'that'
```

That's not much help, a couple of welcomes in there, but no indication of which are keys, and which are values, it's just a listed array dump and, in this case, not particularly helpful. The trick here, is to use a **reference** to the data structure:

```
DB<13> x \%data
0  HASH(0x8194bc4)
   'Hello World' => 'zip'
   'jerry' => 'welcome'
   'this' => 'that'
   'tom' => 'and'
   'welcome' => undef
```

The reference is truly dumped and we can finally see what we're dealing with. Our quoting was perfectly valid but wrong for our purposes, with 'and jerry' being treated as 2 separate words rather than a phrase, thus throwing the evenly paired hash structure out of alignment.

The '**-w**' switch would have told us about this, had we used it at the start, and saved us a lot of trouble:

```
> perl -w data
Odd number of elements in hash assignment at ./data line 5.
```

We fix our quoting: 'tom' = q(and jerry), and run it again, this time we get our expected output:

```
> perl -w data
Hello World
```

While we're here, take a closer look at the '**x**' command, it's really useful and will merrily dump out nested references, complete objects, partial objects – just about whatever you throw at it:

Let's make a quick object and x-plode it, first we'll start the debugger: it wants some form of input from STDIN, so we give it something non-committal, a zero:

```
> perl -de 0
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.
```

Enter `h` or `'h h'` for help, or `'man perldebug'` for more help.

```
main::(-e:1): 0
```

Now build an on-the-fly object over a couple of lines (note the backslash):

```
DB<1> $obj = bless({'unique_id'=>'123', 'attr'=> \
cont: {'col' => 'black', 'things' => [qw(this that etc)]}}, 'MY_class')
```

And let's have a look at it:

```
DB<2> x $obj
0 MY_class=HASH(0x828ad98)
  'attr' => HASH(0x828ad68)
  'col' => 'black'
  'things' => ARRAY(0x828abb8)
    0 'this'
    1 'that'
    2 'etc'
  'unique_id' => 123
DB<3>
```

Useful, huh? You can eval nearly anything in there, and experiment with bits of code or regexes until the cows come home:

```
DB<3> @data = qw(this that the other atheism leather theory scythe)
DB<4> p 'saw -> ' . ($cnt += map { print "\t:\t$_\n" } grep(/the/, sort @data))
atheism
leather
other
scythe
the
theory
saw -> 6
```

If you want to see the command History, type an **'H'**:

```
DB<5> H
4: p 'saw -> ' . ($cnt += map { print "\t:\t$_\n" } grep(/the/, sort @data))
3: @data = qw(this that the other atheism leather theory scythe)
2: x $obj
1: $obj = bless({'unique_id'=>'123', 'attr'=>
{'col' => 'black', 'things' => [qw(this that etc)]}}, 'MY_class')
DB<5>
```

And if you want to repeat any previous command, use the exclamation: **'!'**:

```
DB<5> !4
p 'saw -> ' . ($cnt += map { print "$_\n" } grep(/the/, sort @data))
atheism
leather
other
scythe
the
theory
saw -> 12
```

For more on references see [perlref](#) and [perlreftut](#)

Stepping through code

Here's a simple program which converts between Celsius and Fahrenheit, it too has a problem:

```
#!/usr/bin/perl -w
use strict;

my $arg = $ARGV[0] || '-c20';

if ($arg =~ /^-(c|f)((\-|\+)*\d+(\.\d+)*$)/) {
    my ($deg, $num) = ($1, $2);
    my ($in, $out) = ($num, $num);
    if ($deg eq 'c') {
        $deg = 'f';
        $out = &c2f($num);
    } else {
        $deg = 'c';
        $out = &f2c($num);
    }
    $out = sprintf('%0.2f', $out);
    $out =~ s/^((\-|\+)*\d+)\.0+$/ $1/;
    print "$out $deg\n";
} else {
    print "Usage: $0 -[c|f] num\n";
}
exit;

sub f2c {
    my $f = shift;
    my $c = 5 * $f - 32 / 9;
    return $c;
}

sub c2f {
    my $c = shift;
    my $f = 9 * $c / 5 + 32;
    return $f;
}
```

For some reason, the Fahrenheit to Celsius conversion fails to return the expected output. This is what it does:

```
> temp -c0.72
33.30 f

> temp -f33.3
162.94 c
```

Not very consistent! We'll set a breakpoint in the code manually and run it under the debugger to see what's going on. A breakpoint is a flag, to which the debugger will run without interruption, when it reaches the breakpoint, it will stop execution and offer a prompt for further interaction. In normal use, these debugger commands are completely ignored, and they are safe – if a little messy, to leave in production code.

```
my ($in, $out) = ($num, $num);
$DB::single=2; # insert at line 9!
if ($deg eq 'c')
    ...

> perl -d temp -f33.3
Default die handler restored.
```

Loading DB routines from perl5db.pl version 1.07
 Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

```
main::(temp:4): my $arg = $ARGV[0] || '-c100';
```

We'll simply continue down to our pre-set breakpoint with a 'c':

```
DB<1> c
main::(temp:10): if ($deg eq 'c') {
```

Followed by a view command to see where we are:

```
DB<1> v
7: my ($deg, $num) = ($1, $2);
8: my ($in, $out) = ($num, $num);
9: $DB::single=2;
10==> if ($deg eq 'c') {
11:     $deg = 'f';
12:     $out = &c2f($num);
13: } else {
14:     $deg = 'c';
15:     $out = &f2c($num);
16: }
```

And a print to show what values we're currently using:

```
DB<1> p $deg, $num
f33.3
```

We can put another break point on any line beginning with a colon, we'll use line 17 as that's just as we come out of the subroutine, and we'd like to pause there later on:

```
DB<2> b 17
```

There's no feedback from this, but you can see what breakpoints are set by using the list 'L' command:

```
DB<3> L
temp:
17: print "$out $deg\n";
break if (1)
```

Note that to delete a breakpoint you use 'd' or 'D'.

Now we'll continue down into our subroutine, this time rather than by line number, we'll use the subroutine name, followed by the now familiar 'v':

```
DB<3> c f2c
main::f2c(temp:30): my $f = shift;
DB<4> v
24: exit;
25
26 sub f2c {
27==> my $f = shift;
28: my $c = 5 * $f - 32 / 9;
29: return $c;
30: }
31
32 sub c2f {
33: my $c = shift;
```

Note that if there was a subroutine call between us and line 29, and we wanted to **single-step** through it, we could use the 's' command, and to step over it we would use 'n' which would execute the sub, but not descend into it for inspection. In this case though, we simply continue down to line 29:

```
DB<4> c 29
main::f2c(temp:29):          return $c;
```

And have a look at the return value:

```
DB<5> p $c
162.94444444444444
```

This is not the right answer at all, but the sum looks correct. I wonder if it's anything to do with operator precedence? We'll try a couple of other possibilities with our sum:

```
DB<6> p (5 * $f - 32 / 9)
162.94444444444444
```

```
DB<7> p 5 * $f - (32 / 9)
162.94444444444444
```

```
DB<8> p (5 * $f) - 32 / 9
162.94444444444444
```

```
DB<9> p 5 * ($f - 32) / 9
0.722222222222221
```

:-) that's more like it! Ok, now we can set our return variable and we'll return out of the sub with an 'r':

```
DB<10> $c = 5 * ($f - 32) / 9
DB<11> r
scalar context return from main::f2c: 0.722222222222221
```

Looks good, let's just continue off the end of the script:

```
DB<12> c
0.72 c
Debugged program terminated. Use q to quit or R to restart,
use O inhibit_exit to avoid stopping after program termination,
h q, h R or h O to get additional info.
```

A quick fix to the offending line (insert the missing parentheses) in the actual program and we're finished.

Placeholder for a, w, t, T

Actions, watch variables, stack traces etc.: on the TODO list.

```
a
w
t
T
```

REGULAR EXPRESSIONS

Ever wanted to know what a regex looked like? You'll need perl compiled with the DEBUGGING flag for this one:

```
> perl -Dr -e '/^pe(a)*rl$/i'
Compiling REx `^pe(a)*rl$'
size 17 first at 2
rarest char
at 0
1: BOL(2)
```

```

2: EXACTF <pe>(4)
4: CURLYN[1] {0,32767}(14)
6:  NOTHING(8)
8:  EXACTF <a>(0)
12:  WHILEM(0)
13: NOTHING(14)
14: EXACTF <rl>(16)
16: EOL(17)
17: END(0)
floating ``$ at 4..2147483647 (checking floating) stclass `EXACTF <pe>'
anchored(BOL) minlen 4
Omitting $` $& $' support.
EXECUTING...
Freeing REx: `^pe(a)*rl$'
```

Did you really want to know? :-) For more gory details on getting regular expressions to work, have a look at [perlre](#), [perlretut](#), and to decode the mysterious labels (BOL and CURLYN, etc. above), see [perldebguts](#).

OUTPUT TIPS

To get all the output from your error log, and not miss any messages via helpful operating system buffering, insert a line like this, at the start of your script:

```
$|=1;
```

To watch the tail of a dynamically growing logfile, (from the command line):

```
tail -f $error_log
```

Wrapping all die calls in a handler routine can be useful to see how, and from where, they're being called, [perlvar](#) has more information:

```
BEGIN { $SIG{__DIE__} = sub { require Carp; Carp::confess(@_) } }
```

Various useful techniques for the redirection of STDOUT and STDERR filehandles are explained in [perlopentut](#) and [perlfaq8](#).

CGI

Just a quick hint here for all those CGI programmers who can't figure out how on earth to get past that 'waiting for input' prompt, when running their CGI script from the command-line, try something like this:

```
> perl -d my_cgi.pl -nodebug
```

Of course [CGI](#) and [perlfaq9](#) will tell you more.

GUIs

The command line interface is tightly integrated with an **emacs** extension and there's a **vi** interface too.

You don't have to do this all on the command line, though, there are a few GUI options out there. The nice thing about these is you can wave a mouse over a variable and a dump of its data will appear in an appropriate window, or in a popup balloon, no more tiresome typing of 'x \$varname' :-)

In particular have a hunt around for the following:

ptkdb perlTK based wrapper for the built-in debugger

ddd data display debugger

PerlDevKit and **PerlBuilder** are NT specific

NB. (more info on these and others would be appreciated).

SUMMARY

We've seen how to encourage good coding practices with **use strict** and **-w**. We can run the perl debugger **perl -d scriptname** to inspect your data from within the perl debugger with the **p** and **x** commands. You can walk through your code, set breakpoints with **b** and step through that code with **s** or **n**, continue with **c** and return from a sub with **r**. Fairly intuitive stuff when you get down to it.

There is of course lots more to find out about, this has just scratched the surface. The best way to learn more is to use perldoc to find out more about the language, to read the on-line help ([perldebug](#) is probably the next place to go), and of course, experiment.

SEE ALSO

[perldebug](#), [perldebguts](#), [perldiag](#), [dprofpp](#), [perlrun](#)

AUTHOR

Richard Foley <richard@rfi.net Copyright (c) 2000

CONTRIBUTORS

Various people have made helpful suggestions and contributions, in particular:

Ronald J Kimball <rjk@linguist.dartmouth.edu

Hugo van der Sanden <hv@crypt0.demon.co.uk

Peter Scott <Peter@PSDT.com

NAME

perldebug – Perl debugging

DESCRIPTION

First of all, have you tried using the `-w` switch?

If you're new to the Perl debugger, you may prefer to read [perldebtut](#), which is a tutorial introduction to the debugger.

The Perl Debugger

If you invoke Perl with the `-d` switch, your script runs under the Perl source debugger. This works like an interactive Perl environment, prompting for debugger commands that let you examine source code, set breakpoints, get stack backtraces, change the values of variables, etc. This is so convenient that you often fire up the debugger all by itself just to test out Perl constructs interactively to see what they do. For example:

```
$ perl -d -e 42
```

In Perl, the debugger is not a separate program the way it usually is in the typical compiled environment. Instead, the `-d` flag tells the compiler to insert source information into the parse trees it's about to hand off to the interpreter. That means your code must first compile correctly for the debugger to work on it. Then when the interpreter starts up, it preloads a special Perl library file containing the debugger.

The program will halt *right before* the first run-time executable statement (but see below regarding compile-time statements) and ask you to enter a debugger command. Contrary to popular expectations, whenever the debugger halts and shows you a line of code, it always displays the line it's *about* to execute, rather than the one it has just executed.

Any command not recognized by the debugger is directly executed (`eval'd`) as Perl code in the current package. (The debugger uses the DB package for keeping its own state information.)

For any text entered at the debugger prompt, leading and trailing whitespace is first stripped before further processing. If a debugger command coincides with some function in your own program, merely precede the function with something that doesn't look like a debugger command, such as a leading `;` or perhaps a `+`, or by wrapping it with parentheses or braces.

Debugger Commands

The debugger understands the following commands:

h Prints out a summary help message

h [command] Prints out a help message for the given debugger command.

h h The special argument of `h h` produces the entire help page, which is quite long.

If the output of the `h h` command (or any command, for that matter) scrolls past your screen, precede the command with a leading pipe symbol so that it's run through your pager, as in

```
DB> |h h
```

You may change the pager which is used via `o pager=...` command.

p expr Same as `print { $DB::OUT } expr` in the current package. In particular, because this is just Perl's own `print` function, this means that nested data structures and objects are not dumped, unlike with the `x` command.

The `DB::OUT` filehandle is opened to `/dev/tty`, regardless of where `STDOUT` may be redirected to.

<code>x [maxdepth] expr</code>	<p>Evaluates its expression in list context and dumps out the result in a pretty-printed fashion. Nested data structures are printed out recursively, unlike the real <code>print</code> function in Perl. When dumping hashes, you'll probably prefer <code>'x \%h'</code> rather than <code>'x %h'</code>. See Dumpvalue if you'd like to do this yourself.</p> <p>The output format is governed by multiple options described under "Configurable Options".</p> <p>If the <code>maxdepth</code> is included, it must be a numeral <i>N</i>; the value is dumped only <i>N</i> levels deep, as if the <code>dumpDepth</code> option had been temporarily set to <i>N</i>.</p>
<code>V [pkg [vars]]</code>	<p>Display all (or some) variables in package (defaulting to <code>main</code>) using a data pretty-printer (hashes show their keys and values so you see what's what, control characters are made printable, etc.). Make sure you don't put the type specifier (like <code>\$</code>) there, just the symbol names, like this:</p> <pre>V DB filename line</pre> <p>Use <code>~pattern</code> and <code>!pattern</code> for positive and negative regexes.</p> <p>This is similar to calling the <code>x</code> command on each applicable var.</p>
<code>X [vars]</code>	Same as <code>V currentpackage [vars]</code> .
<code>y [level [vars]]</code>	<p>Display all (or some) lexical variables (mnemonic: <code>mY</code> variables) in the current scope or <i>level</i> scopes higher. You can limit the variables that you see with <i>vars</i> which works exactly as it does for the <code>V</code> and <code>X</code> commands. Requires the <code>PadWalker</code> module version 0.08 or higher; will warn if this isn't installed. Output is pretty-printed in the same style as for <code>V</code> and the format is controlled by the same options.</p>
<code>T</code>	Produce a stack backtrace. See below for details on its output.
<code>s [expr]</code>	Single step. Executes until the beginning of another statement, descending into subroutine calls. If an expression is supplied that includes function calls, it too will be single-stepped.
<code>n [expr]</code>	Next. Executes over subroutine calls, until the beginning of the next statement. If an expression is supplied that includes function calls, those functions will be executed with stops before each statement.
<code>r</code>	Continue until the return from the current subroutine. Dump the return value if the <code>PrintRet</code> option is set (default).
<code><CR</code>	Repeat last <code>n</code> or <code>s</code> command.
<code>c [line sub]</code>	Continue, optionally inserting a one-time-only breakpoint at the specified line or subroutine.
<code>l</code>	List next window of lines.
<code>l min+incr</code>	List <code>incr+1</code> lines starting at <code>min</code> .
<code>l min-max</code>	List lines <code>min</code> through <code>max</code> . <code>l -</code> is synonymous to <code>-</code> .
<code>l line</code>	List a single line.
<code>l subname</code>	List first window of lines from subroutine. <i>subname</i> may be a variable that contains a code reference.
<code>-</code>	List previous window of lines.

v [line]	View a few lines of code around the current line.
.	Return the internal debugger pointer to the line last executed, and print out that line.
f filename	Switch to viewing a different file or eval statement. If <i>filename</i> is not a full pathname found in the values of %INC, it is considered a regex. eval'd strings (when accessible) are considered to be filenames: f (eval 7) and f eval 7\b access the body of the 7th eval'd string (in the order of execution). The bodies of the currently executed eval and of eval'd strings that define subroutines are saved and thus accessible.
/pattern/	Search forwards for pattern (a Perl regex); final / is optional. The search is case-insensitive by default.
?pattern?	Search backwards for pattern; final ? is optional. The search is case-insensitive by default.
L [abw]	List (default all) actions, breakpoints and watch expressions
S [[!]regex]	List subroutine names [not] matching the regex.
t	Toggle trace mode (see also the AutoTrace option).
t expr	Trace through execution of expr. See Frame Listing Output Examples in perldebguts for examples.
b	Sets breakpoint on current line
b [line] [condition]	Set a breakpoint before the given line. If a condition is specified, it's evaluated each time the statement is reached: a breakpoint is taken only if the condition is true. Breakpoints may only be set on lines that begin an executable statement. Conditions don't use if: <pre> b 237 \$x > 30 b 237 ++\$count237 < 11 b 33 /pattern/i </pre>
b subname [condition]	Set a breakpoint before the first line of the named subroutine. <i>subname</i> may be a variable containing a code reference (in this case <i>condition</i> is not supported).
b postpone subname [condition]	Set a breakpoint at first line of subroutine after it is compiled.
b load filename	Set a breakpoint before the first executed line of the <i>filename</i> , which should be a full pathname found amongst the %INC values.
b compile subname	Sets a breakpoint before the first statement executed after the specified subroutine is compiled.
B line	Delete a breakpoint from the specified <i>line</i> .
B *	Delete all installed breakpoints.
a [line] command	Set an action to be done before the line is executed. If <i>line</i> is omitted, set an action on the line about to be executed. The sequence of steps taken by the debugger is <pre> 1. check for a breakpoint at this line 2. print the line if necessary (tracing) 3. do any actions associated with that line </pre>

4. prompt user if at a breakpoint or in single-step
5. evaluate line

For example, this will print out \$foo every time line 53 is passed:

```
a 53 print "DB FOUND $foo\n"
```

A line	Delete an action from the specified line.
A *	Delete all installed actions.
w expr	Add a global watch-expression. We hope you know what one of these is, because they're supposed to be obvious.
W expr	Delete watch-expression
W *	Delete all watch-expressions.
o	Display all options
o boolooption ...	Set each listed Boolean option to the value 1.
o anyoption? ...	Print out the value of one or more options.
o option=value ...	Set the value of one or more options. If the value has internal whitespace, it should be quoted. For example, you could set <code>o pager="less -MQeicsNfr"</code> to call less with those specific options. You may use either single or double quotes, but if you do, you must escape any embedded instances of same sort of quote you began with, as well as any escaping any escapes that immediately precede that quote but which are not meant to escape the quote itself. In other words, you follow single-quoting rules irrespective of the quote; eg: <code>o option='this isn\'t bad'</code> or <code>o option="She said, \"Isn't it?\""</code> . For historical reasons, the <code>=value</code> is optional, but defaults to 1 only where it is safe to do so—that is, mostly for Boolean options. It is always better to assign a specific value using <code>=</code> . The <code>option</code> can be abbreviated, but for clarity probably should not be. Several options can be set together. See " Configurable Options " for a list of these.
< ?	List out all pre-prompt Perl command actions.
< [command]	Set an action (Perl command) to happen before every debugger prompt. A multi-line command may be entered by backslashing the newlines. WARNING If <code>command</code> is missing, all actions are wiped out!
<< command	Add an action (Perl command) to happen before every debugger prompt. A multi-line command may be entered by backwhacking the newlines.
?	List out post-prompt Perl command actions.
command	Set an action (Perl command) to happen after the prompt when you've just given a command to return to executing the script. A multi-line command may be entered by backslashing the newlines (we bet you couldn't've guessed this by now). WARNING If <code>command</code> is missing, all actions are wiped out!
command	Adds an action (Perl command) to happen after the prompt when you've just given a command to return to executing the script. A multi-line command may be entered by backslashing the newlines.
{ ?	List out pre-prompt debugger commands.

- `{ [command]` Set an action (debugger command) to happen before every debugger prompt. A multi-line command may be entered in the customary fashion. **WARNING** If `command` is missing, all actions are wiped out!
- Because this command is in some senses new, a warning is issued if you appear to have accidentally entered a block instead. If that's what you mean to do, write it as with `; { ... }` or even `do { ... }`.
- `{{ command` Add an action (debugger command) to happen before every debugger prompt. A multi-line command may be entered, if you can guess how: see above.
- `! number` Redo a previous command (defaults to the previous command).
- `! -number` Redo number'th previous command.
- `! pattern` Redo last command that started with pattern. See `o recallCommand`, too.
- `!! cmd` Run `cmd` in a subprocess (reads from `DB::IN`, writes to `DB::OUT`) See `o shellBang`, also. Note that the user's current shell (well, their `$ENV{SHELL}` variable) will be used, which can interfere with proper interpretation of exit status or signal and coredump information.
- `source file` Read and execute debugger commands from *file*. *file* may itself contain source commands.
- `H -number` Display last *n* commands. Only commands longer than one character are listed. If *number* is omitted, list them all.
- `q` or `^D` Quit. ("quit" doesn't work for this, unless you've made an alias) This is the only supported way to exit the debugger, though typing `exit` twice might work.
- Set the `inhibit_exit` option to 0 if you want to be able to step off the end the script. You may also need to set `$finished` to 0 if you want to step through global destruction.
- `R` Restart the debugger by `exec()`ing a new session. We try to maintain your history across this, but internal settings and command-line options may be lost.
- The following setting are currently preserved: history, breakpoints, actions, debugger options, and the Perl command-line options `-w`, `-I`, and `-e`.
- `|dbcmd` Run the debugger command, piping `DB::OUT` into your current pager.
- `||dbcmd` Same as `|dbcmd` but `DB::OUT` is temporarily selected as well.
- `= [alias value]` Define a command alias, like
- `= quit q`
- or list current aliases.
- `command` Execute `command` as a Perl statement. A trailing semicolon will be supplied. If the Perl statement would otherwise be confused for a Perl debugger, use a leading semicolon, too.
- `m expr` List which methods may be called on the result of the evaluated expression. The expression may evaluated to a reference to a blessed object, or to a package name.
- `M` Displays all loaded modules and their versions
- `man [manpage]` Despite its name, this calls your system's default documentation viewer on the given page, or on the viewer itself if *manpage* is omitted. If that viewer is **man**, the current Config information is used to invoke **man** using the proper `MANPATH` or `-M manpath` option. Failed lookups of the form `XXX` that match known manpages of the form *perlXXX* will be

retrieved. This lets you type `man debug` or `man op` from the debugger.

On systems traditionally bereft of a usable **man** command, the debugger invokes **perldoc**. Occasionally this determination is incorrect due to recalcitrant vendors or rather more felicitously, to enterprising users. If you fall into either category, just manually set the `$DB::doccmd` variable to whatever viewer to view the Perl documentation on your system. This may be set in an rc file, or through direct assignment. We're still waiting for a working example of something along the lines of:

```
$DB::doccmd = 'netscape -remote http://something.here/';
```

Configurable Options

The debugger has numerous options settable using the `o` command, either interactively or from the environment or an rc file. (`./perl.db` or `~/perl.db` under Unix.)

`recallCommand, ShellBang`

The characters used to recall command or spawn shell. By default, both are set to `!`, which is unfortunate.

`pager`

Program to use for output of pager-piped commands (those beginning with a `|` character.) By default, `$ENV{PAGER}` will be used. Because the debugger uses your current terminal characteristics for bold and underlining, if the chosen pager does not pass escape sequences through unchanged, the output of some debugger commands will not be readable when sent through the pager.

`tkRunning`

Run Tk while prompting (with `ReadLine`).

`signalLevel, warnLevel, dieLevel`

Level of verbosity. By default, the debugger leaves your exceptions and warnings alone, because altering them can break correctly running programs. It will attempt to print a message when uncaught INT, BUS, or SEGV signals arrive. (But see the mention of signals in *BUGS* below.)

To disable this default safe mode, set these values to something higher than 0. At a level of 1, you get backtraces upon receiving any kind of warning (this is often annoying) or exception (this is often valuable). Unfortunately, the debugger cannot discern fatal exceptions from non-fatal ones. If `dieLevel` is even 1, then your non-fatal exceptions are also traced and unceremoniously altered if they came from `eval 'd` strings or from any kind of `eval` within modules you're attempting to load. If `dieLevel` is 2, the debugger doesn't care where they came from: It usurps your exception handler and prints out a trace, then modifies all exceptions with its own embellishments. This may perhaps be useful for some tracing purposes, but tends to hopelessly destroy any program that takes its exception handling seriously.

`AutoTrace`

Trace mode (similar to `t` command, but can be put into `PERLDB_OPTS`).

`LineInfo`

File or pipe to print line number info to. If it is a pipe (say, `|visual_perl_db`), then a short message is used. This is the mechanism used to interact with a slave editor or visual debugger, such as the special `vi` or `emacs` hooks, or the `ddd` graphical debugger.

`inhibit_exit`

If 0, allows *stepping off* the end of the script.

`PrintRet`

Print return value after `r` command if set (default).

`ornaments`

Affects screen appearance of the command line (see *Term::ReadLine*). There is currently no way to disable these, which can render some output illegible on some displays, or with some pagers. This is considered a bug.

- frame** Affects the printing of messages upon entry and exit from subroutines. If `frame & 2` is false, messages are printed on entry only. (Printing on exit might be useful if interspersed with other messages.)
- If `frame & 4`, arguments to functions are printed, plus context and caller info. If `frame & 8`, overloaded `stringify` and `tied FETCH` is enabled on the printed arguments. If `frame & 16`, the return value from the subroutine is printed.
- The length at which the argument list is truncated is governed by the next option:
- maxTraceLen** Length to truncate the argument list when the `frame` option's bit 4 is set.
- windowSize** Change the size of code list window (default is 10 lines).
- The following options affect what happens with `V`, `X`, and `x` commands:
- arrayDepth**, **hashDepth**
Print only first `N` elements ('' for all).
- dumpDepth** Limit recursion depth to `N` levels when dumping structures. Negative values are interpreted as infinity. Default: infinity.
- compactDump**, **veryCompact**
Change the style of array and hash output. If `compactDump`, short array may be printed on one line.
- globPrint** Whether to print contents of globs.
- DumpDBFiles** Dump arrays holding debugged files.
- DumpPackages**
Dump symbol tables of packages.
- DumpReused** Dump contents of "reused" addresses.
- quote**, **HighBit**, **undefPrint**
Change the style of string dump. The default value for `quote` is `auto`; one can enable double-quotish or single-quotish format by setting it to `"` or `'`, respectively. By default, characters with their high bit set are printed verbatim.
- UsageOnly** Rudimentary per-package memory usage dump. Calculates total size of strings found in variables in the package. This does not include lexicals in a module's file scope, or lost in closures.

After the `rc` file is read, the debugger reads the `$ENV{PERLDB_OPTS}` environment variable and parses this as the remainder of a `'O ...'` line as one might enter at the debugger prompt. You may place the initialization options `TTY`, `noTTY`, `ReadLine`, and `NonStop` there.

If your `rc` file contains:

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace");
```

then your script will run without human intervention, putting trace information into the file `db.out`. (If you interrupt it, you'd better reset `LineInfo` to `/dev/tty` if you expect to see anything.)

- TTY** The TTY to use for debugging I/O.
- noTTY** If set, the debugger goes into `NonStop` mode and will not connect to a TTY. If interrupted (or if control goes to the debugger via explicit setting of `$DB::signal` or `$DB::single` from the Perl script), it connects to a TTY specified in the `TTY` option at startup, or to a `tty` found at runtime using the `Term::Rendezvous` module of your choice.

This module should implement a method named `new` that returns an object with two methods: `IN` and `OUT`. These should return filehandles to use for debugging input and output correspondingly. The `new` method should inspect an argument containing the value of `$ENV{PERLDB_NOTTY}` at startup, or `"/tmp/perldebugtty$$"` otherwise. This file is not inspected for proper ownership, so security hazards are theoretically possible.

ReadLine If false, readline support in the debugger is disabled in order to debug applications that themselves use `ReadLine`.

NonStop If set, the debugger goes into non-interactive mode until interrupted, or programmatically by setting `$DB::signal` or `$DB::single`.

Here's an example of using the `$ENV{PERLDB_OPTS}` variable:

```
$ PERLDB_OPTS="NonStop frame=2" perl -d myprogram
```

That will run the script **myprogram** without human intervention, printing out the call tree with entry and exit points. Note that `NonStop=1 frame=2` is equivalent to `N f=2`, and that originally, options could be uniquely abbreviated by the first letter (modulo the `Dump*` options). It is nevertheless recommended that you always spell them out in full for legibility and future compatibility.

Other examples include

```
$ PERLDB_OPTS="NonStop LineInfo=listing frame=2" perl -d myprogram
```

which runs script non-interactively, printing info on each entry into a subroutine and each executed line into the file named *listing*. (If you interrupt it, you would better reset `LineInfo` to something "interactive!")

Other examples include (using standard shell syntax to show environment variable settings):

```
$ ( PERLDB_OPTS="NonStop frame=1 AutoTrace LineInfo=tperl.out"
  perl -d myprogram )
```

which may be useful for debugging a program that uses `Term::ReadLine` itself. Do not forget to detach your shell from the TTY in the window that corresponds to `/dev/ttyXX`, say, by issuing a command like

```
$ sleep 1000000
```

See [Debugger Internals in perldebugs](#) for details.

Debugger input/output

Prompt The debugger prompt is something like

```
DB<8>
```

or even

```
DB<<17>>
```

where that number is the command number, and which you'd use to access with the built-in **cs**h-like history mechanism. For example, `!17` would repeat command number 17. The depth of the angle brackets indicates the nesting depth of the debugger. You could get more than one set of brackets, for example, if you'd already at a breakpoint and then printed the result of a function call that itself has a breakpoint, or you step into an expression via `s/n/t` expression command.

Multiline commands

If you want to enter a multi-line command, such as a subroutine definition with several statements or a format, escape the newline that would normally end the debugger command with a backslash. Here's an example:

```
DB<1> for (1..4) {          \
cont:    print "ok\n";      \
cont: }
```



```
ok
ok
ok
ok
```

Note that this business of escaping a newline is specific to interactive commands typed into the debugger.

Stack backtrace

Here's an example of what a stack backtrace via T command might look like:

```
$ = main::infested called from file 'Ambulation.pm' line 10
@ = Ambulation::legs(1, 2, 3, 4) called from file 'camel_flea' line 7
$ = main::pests('bactrian', 4) called from file 'camel_flea' line 4
```

The left-hand character up there indicates the context in which the function was called, with \$ and @ meaning scalar or list contexts respectively, and . meaning void context (which is actually a sort of scalar context). The display above says that you were in the function `main::infested` when you ran the stack dump, and that it was called in scalar context from line 10 of the file *Ambulation.pm*, but without any arguments at all, meaning it was called as `&infested`. The next stack frame shows that the function `Ambulation::legs` was called in list context from the *camel_flea* file with four arguments. The last stack frame shows that `main::pests` was called in scalar context, also from *camel_flea*, but from line 4.

If you execute the T command from inside an active use statement, the backtrace will contain both a `require` frame and an `eval` frame.

Line Listing Format

This shows the sorts of output the l command can produce:

```
DB<<13>> l
101:                @i{@i} = ();
102:b                @isa{@i,$pack} = ()
103                  if(exists $i{$prevpack} || exists $isa{$pack});
104                  }
105
106                  next
107==>                if(exists $isa{$pack});
108
109:a                if ($extra-- > 0) {
110:                  %isa = ($pack,1);
```

Breakable lines are marked with `..`. Lines with breakpoints are marked by `b` and those with actions by `a`. The line that's about to be executed is marked by `< ==`.

Please be aware that code in debugger listings may not look the same as your original source code. Line directives and external source filters can alter the code before Perl sees it, causing code to move from its original positions or take on entirely different forms.

Frame listing

When the frame option is set, the debugger would print entered (and optionally exited) subroutines in different styles. See [perldebbugs](#) for incredibly long examples of these.

Debugging compile-time statements

If you have compile-time executable statements (such as code within BEGIN and CHECK blocks or use statements), these will *not* be stopped by debugger, although `requires` and `INIT` blocks will, and compile-time statements can be traced with `AutoTrace` option set in `PERLDB_OPTS`). From your own Perl code, however, you can transfer control back to the debugger using the following statement, which is harmless if the debugger is not running:

```
$DB::single = 1;
```

If you set `$DB::single` to 2, it's equivalent to having just typed the `n` command, whereas a value of 1 means the `s` command. The `$DB::trace` variable should be set to 1 to simulate having typed the `t` command.

Another way to debug compile-time code is to start the debugger, set a breakpoint on the *load* of some module:

```
DB<7> b load f:/perl/lib/lib/Carp.pm
Will stop on load of 'f:/perl/lib/lib/Carp.pm'.
```

and then restart the debugger using the `R` command (if possible). One can use `b compile subname` for the same purpose.

Debugger Customization

The debugger probably contains enough configuration hooks that you won't ever have to modify it yourself. You may change the behaviour of debugger from within the debugger using its `o` command, from the command line via the `PERLDB_OPTS` environment variable, and from customization files.

You can do some customization by setting up a *.perldb* file, which contains initialization code. For instance, you could make aliases like these (the last one is one people expect to be there):

```
$DB::alias{'len'} = 's/^len(.*)/p length($1)/';
$DB::alias{'stop'} = 's/^stop (at|in)/b/';
$DB::alias{'ps'} = 's/^ps\b/p scalar /';
$DB::alias{'quit'} = 's/^quit(\s*)/exit/';
```

You can change options from *.perldb* by using calls like this one;

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace=1 frame=2");
```

The code is executed in the package `DB`. Note that *.perldb* is processed before processing `PERLDB_OPTS`. If *.perldb* defines the subroutine *afterinit*, that function is called after debugger initialization ends. *.perldb* may be contained in the current directory, or in the home directory. Because this file is sourced in by Perl and may contain arbitrary commands, for security reasons, it must be owned by the superuser or the current user, and writable by no one but its owner.

If you want to modify the debugger, copy *perl5db.pl* from the Perl library to another name and hack it to your heart's content. You'll then want to set your `PERL5DB` environment variable to say something like this:

```
BEGIN { require "myperl5db.pl" }
```

As a last resort, you could also use `PERL5DB` to customize the debugger by directly setting internal variables or calling debugger functions.

Note that any variables and functions that are not documented in this document (or in *perldebguts*) are considered for internal use only, and as such are subject to change without notice.

Readline Support

As shipped, the only command-line history supplied is a simplistic one that checks for leading exclamation points. However, if you install the `Term::ReadKey` and `Term::ReadLine` modules from CPAN, you will have full editing capabilities much like GNU *readline*(3) provides. Look for these in the *modules/by-module/Term* directory on CPAN. These do not support normal *vi* command-line editing, however.

A rudimentary command-line completion is also available. Unfortunately, the names of lexical variables are not available for completion.

Editor Support for Debugging

If you have the FSF's version of **emacs** installed on your system, it can interact with the Perl debugger to provide an integrated software development environment reminiscent of its interactions with C debuggers.

Perl comes with a start file for making **emacs** act like a syntax-directed editor that understands (some of) Perl's syntax. Look in the *emacs* directory of the Perl source distribution.

A similar setup by Tom Christiansen for interacting with any vendor-shipped **vi** and the X11 window system is also available. This works similarly to the integrated multiwindow support that **emacs** provides, where the debugger drives the editor. At the time of this writing, however, that tool's eventual location in the Perl distribution was uncertain.

Users of **vi** should also look into **vim** and **gvim**, the mousey and windy version, for coloring of Perl keywords.

Note that only perl can truly parse Perl, so all such CASE tools fall somewhat short of the mark, especially if you don't program your Perl as a C programmer might.

The Perl Profiler

If you wish to supply an alternative debugger for Perl to run, just invoke your script with a colon and a package argument given to the **-d** flag. The most popular alternative debuggers for Perl is the Perl profiler. `Devel::DProf` is now included with the standard Perl distribution. To profile your Perl program in the file *mycode.pl*, just type:

```
$ perl -d:DProf mycode.pl
```

When the script terminates the profiler will dump the profile information to a file called *tmon.out*. A tool like **dprofpp**, also supplied with the standard Perl distribution, can be used to interpret the information in that profile.

Debugging regular expressions

use `re 'debug'` enables you to see the gory details of how the Perl regular expression engine works. In order to understand this typically voluminous output, one must not only have some idea about how regular expression matching works in general, but also know how Perl's regular expressions are internally compiled into an automaton. These matters are explored in some detail in

[Debugging regular expressions in perldebugs](#).

Debugging memory usage

Perl contains internal support for reporting its own memory usage, but this is a fairly advanced concept that requires some understanding of how memory allocation works. See

[Debugging Perl memory usage in perldebugs](#) for the details.

SEE ALSO

You did try the **-w** switch, didn't you?

[perldebtut](#), [perldebugs](#), [re](#), [DB](#), [Devel::DProf](#), [dprofpp](#), [Dumpvalue](#), and [perlrun](#).

BUGS

You cannot get stack frame information or in any fashion debug functions that were not compiled by Perl, such as those from C or C++ extensions.

If you alter your `@_` arguments in a subroutine (such as with `shift` or `pop`), the stack backtrace will not show the original values.

The debugger does not currently work in conjunction with the **-W** command-line switch, because it itself is not free of warnings.

If you're in a slow syscall (like waiting, accepting, or reading from your keyboard or a socket) and haven't set up your own `$SIG{INT}` handler, then you won't be able to CTRL-C your way back to the debugger, because the debugger's own `$SIG{INT}` handler doesn't understand that it needs to raise an

exception to `longjmp(3)` out of slow syscalls.

NAME

perldelta – what is new for perl v5.8.0

DESCRIPTION

This document describes differences between the 5.6.0 release and the 5.8.0 release.

Many of the bug fixes in 5.8.0 were already seen in the 5.6.1 maintenance release since the two releases were kept closely coordinated (while 5.8.0 was still called 5.7.something).

Changes that were integrated into the 5.6.1 release are marked [561]. Many of these changes have been further developed since 5.6.1 was released, those are marked [561+].

You can see the list of changes in the 5.6.1 release (both from the 5.005_03 release and the 5.6.0 release) by reading [perl561delta](#).

Highlights In 5.8.0

- Better Unicode support
- New IO Implementation
- New Thread Implementation
- Better Numeric Accuracy
- Safe Signals
- Many New Modules
- More Extensive Regression Testing

Incompatible Changes

Binary Incompatibility

Perl 5.8 is not binary compatible with earlier releases of Perl.

You have to recompile your XS modules.

(Pure Perl modules should continue to work.)

The major reason for the discontinuity is the new IO architecture called PerlIO. PerlIO is the default configuration because without it many new features of Perl 5.8 cannot be used. In other words: you just have to recompile your modules containing XS code, sorry about that.

In future releases of Perl, non-PerlIO aware XS modules may become completely unsupported. This shouldn't be too difficult for module authors, however: PerlIO has been designed as a drop-in replacement (at the source code level) for the stdio interface.

Depending on your platform, there are also other reasons why we decided to break binary compatibility, please read on.

64-bit platforms and malloc

If your pointers are 64 bits wide, the Perl malloc is no longer being used because it does not work well with 8-byte pointers. Also, usually the system mallocs on such platforms are much better optimized for such large memory models than the Perl malloc. Some memory-hungry Perl applications like the PDL don't work well with Perl's malloc. Finally, other applications than Perl (such as mod_perl) tend to prefer the system malloc. Such platforms include Alpha and 64-bit HPPA, MIPS, PPC, and Sparc.

AIX Dynaloading

The AIX dynaloading now uses in AIX releases 4.3 and newer the native dlopen interface of AIX instead of the old emulated interface. This change will probably break backward compatibility with compiled modules.

The change was made to make Perl more compliant with other applications like mod_perl which are using the AIX native interface.

Attributes for `my` variables now handled at run-time

The `my EXPR : ATTRS` syntax now applies variable attributes at run-time. (Subroutine and `our` variables still get attributes applied at compile-time.) See [attributes](#) for additional details. In particular, however, this allows variable attributes to be useful for `tie` interfaces, which was a deficiency of earlier releases. Note that the new semantics doesn't work with the `Attribute::Handlers` module (as of version 0.76).

Socket Extension Dynamic in VMS

The Socket extension is now dynamically loaded instead of being statically built in. This may or may not be a problem with ancient TCP/IP stacks of VMS: we do not know since we weren't able to test Perl in such configurations.

IEEE-format Floating Point Default on OpenVMS Alpha

Perl now uses IEEE format (`T_FLOAT`) as the default internal floating point format on OpenVMS Alpha, potentially breaking binary compatibility with external libraries or existing data. `G_FLOAT` is still available as a configuration option. The default on VAX (`D_FLOAT`) has not changed.

New Unicode Semantics (no more `use utf8`, almost)

Previously in Perl 5.6 to use Unicode one would say "use utf8" and then the operations (like string concatenation) were Unicode-aware in that lexical scope.

This was found to be an inconvenient interface, and in Perl 5.8 the Unicode model has completely changed: now the "Unicodeness" is bound to the data itself, and for most of the time "use utf8" is not needed at all. The only remaining use of "use utf8" is when the Perl script itself has been written in the UTF-8 encoding of Unicode. (UTF-8 has not been made the default since there are many Perl scripts out there that are using various national eight-bit character sets, which would be illegal in UTF-8.)

See [perluniintro](#) for the explanation of the current model, and [utf8](#) for the current use of the utf8 pragma.

New Unicode Properties

Unicode *scripts* are now supported. Scripts are similar to (and superior to) Unicode *blocks*. The difference between scripts and blocks is that scripts are the glyphs used by a language or a group of languages, while the blocks are more artificial groupings of (mostly) 256 characters based on the Unicode numbering.

In general, scripts are more inclusive, but not universally so. For example, while the script `Latin` includes all the Latin characters and their various diacritic-adorned versions, it does not include the various punctuation or digits (since they are not solely Latin).

A number of other properties are now supported, including `\p{L&}`, `\p{Any}` `\p{Assigned}`, `\p{Unassigned}`, `\p{Blank}` [561] and `\p{SpacePerl}` [561] (along with their `\P{...}` versions, of course). See [perlunicode](#) for details, and more additions.

The `In` or `Is` prefix to names used with the `\p{...}` and `\P{...}` are now almost always optional. The only exception is that a `In` prefix is required to signify a Unicode block when a block name conflicts with a script name. For example, `\p{Tibetan}` refers to the script, while `\p{InTibetan}` refers to the block. When there is no name conflict, you can omit the `In` from the block name (e.g. `\p{BraillePatterns}`), but to be safe, it's probably best to always use the `In`).

REF(...) Instead Of SCALAR(...)

A reference to a reference now stringifies as "REF(0x81485ec)" instead of "SCALAR(0x81485ec)" in order to be more consistent with the return value of `ref()`.

pack/unpack D/F recycled

The undocumented `pack/unpack` template letters `D/F` have been recycled for better use: now they stand for long double (if supported by the platform) and `NV` (Perl internal floating point type). (They used to be aliases for `d/f`, but you never knew that.)

glob() now returns filenames in alphabetical order

The list of filenames from `glob()` (or `<...>`) is now by default sorted alphabetically to be csh-compliant (which is what happened before in most UNIX platforms). (`bsd_glob()` does still sort platform natively, ASCII or EBCDIC, unless `GLOB_ALPHASORT` is specified.) [561]

Deprecations

- The semantics of `bless(REF, REF)` were unclear and until someone proves it to make some sense, it is forbidden.
- The obsolete `chat2` library that should never have been allowed to escape the laboratory has been decommissioned.
- Using `chdir("")` or `chdir(undef)` instead of explicit `chdir()` is doubtful. A failure (think `chdir(some_function())`) can lead into unintended `chdir()` to the home directory, therefore this behaviour is deprecated.
- The builtin `dump()` function has probably outlived most of its usefulness. The core-dumping functionality will remain in future available as an explicit call to `CORE::dump()`, but in future releases the behaviour of an unqualified `dump()` call may change.
- The very dusty examples in the `eg/` directory have been removed. Suggestions for new shiny examples welcome but the main issue is that the examples need to be documented, tested and (most importantly) maintained.
- The (bogus) escape sequences `\8` and `\9` now give an optional warning ("Unrecognized escape passed through"). There is no need to `\-escape` any `\w` character.
- The `*glob{FILEHANDLE}` is deprecated, use `*glob{IO}` instead.
- The `package;` syntax (`package` without an argument) has been deprecated. Its semantics were never that clear and its implementation even less so. If you have used that feature to disallow all but fully qualified variables, use `strict;` instead.
- The unimplemented POSIX regex features `[[.cc.]]` and `[[=c=]]` are still recognised but now cause fatal errors. The previous behaviour of ignoring them by default and warning if requested was unacceptable since it, in a way, falsely promised that the features could be used.
- In future releases, non-PerlIO aware XS modules may become completely unsupported. Since PerlIO is a drop-in replacement for `stdio` at the source code level, this shouldn't be that drastic a change.
- Previous versions of perl and some readings of some sections of Camel III implied that the `:raw` "discipline" was the inverse of `:crlf`. Turning off `"clrfness"` is no longer enough to make a stream truly binary. So the PerlIO `:raw` layer (or "discipline", to use the Camel book's older terminology) is now formally defined as being equivalent to `binmode(FH)` – which is in turn defined as doing whatever is necessary to pass each byte as-is without any translation. In particular `binmode(FH)` – and hence `:raw` – will now turn off both CRLF and UTF-8 translation and remove other layers (e.g. `:encoding()`) which would modify byte stream.
- The current user-visible implementation of pseudo-hashes (the weird use of the first array element) is deprecated starting from Perl 5.8.0 and will be removed in Perl 5.10.0, and the feature will be implemented differently. Not only is the current interface rather ugly, but the current implementation slows down normal array and hash use quite noticeably. The `fields` pragma interface will remain available. The *restricted hashes* interface is expected to be the replacement interface (see [Hash::Util](#)). If your existing programs depends on the underlying implementation, consider using [Class::PseudoHash](#) from CPAN.
- The syntaxes `< @a-[...]` and `< %h-{...}` have now been deprecated.

- After years of trying, `suidperl` is considered to be too complex to ever be considered truly secure. The `suidperl` functionality is likely to be removed in a future release.
- The 5.005 threads model (module `Thread`) is deprecated and expected to be removed in Perl 5.10. Multithreaded code should be migrated to the new `ithreads` model (see [threads](#), [threads::shared](#) and [perlthrtut](#)).
- The long deprecated uppercase aliases for the string comparison operators (`EQ`, `NE`, `LT`, `LE`, `GE`, `GT`) have now been removed.
- The `tr//C` and `tr//U` features have been removed and will not return; the interface was a mistake. Sorry about that. For similar functionality, see `pack('U0', ...)` and `pack('C0', ...)`. [561]
- Earlier Perls treated `"sub foo (@bar)"` as equivalent to `"sub foo (@)"`. The prototypes are now checked better at compile-time for invalid syntax. An optional warning is generated ("Illegal character in prototype...") but this may be upgraded to a fatal error in a future release.
- The `exec LIST` and `system LIST` operations now produce warnings on tainted data and in some future release they will produce fatal errors.
- The existing behaviour when localising tied arrays and hashes is wrong, and will be changed in a future release, so do not rely on the existing behaviour. See ["Localising Tied Arrays and Hashes Is Broken"](#).

Core Enhancements

Unicode Overhaul

Unicode in general should be now much more usable than in Perl 5.6.0 (or even in 5.6.1). Unicode can be used in hash keys, Unicode in regular expressions should work now, Unicode in `tr//` should work now, Unicode in I/O should work now. See [perluniintro](#) for introduction and [perlunicode](#) for details.

- The Unicode Character Database coming with Perl has been upgraded to Unicode 3.2.0. For more information, see <http://www.unicode.org/>. [561+] (5.6.1 has UCD 3.0.1.)
- For developers interested in enhancing Perl's Unicode capabilities: almost all the UCD files are included with the Perl distribution in the *lib/unicore* subdirectory. The most notable omission, for space considerations, is the UniHan database.
- The properties `\p{Blank}` and `\p{SpacePerl}` have been added. "Blank" is like `C isblank()`, that is, it contains only "horizontal whitespace" (the space character is, the newline isn't), and the "SpacePerl" is the Unicode equivalent of `\s` (`\p{Space}` isn't, since that includes the vertical tabulator character, whereas `\s` doesn't.)

See "New Unicode Properties" earlier in this document for additional information on changes with Unicode properties.

PerlIO is Now The Default

- IO is now by default done via PerlIO rather than system's "stdio". PerlIO allows "layers" to be "pushed" onto a file handle to alter the handle's behaviour. Layers can be specified at open time via 3-arg form of open:

```
open($fh, '>:crlf :utf8', $path) || ...
```

or on already opened handles via extended binmode:

```
binmode($fh, ':encoding(iso-8859-7)');
```

The built-in layers are: `unix` (low level read/write), `stdio` (as in previous Perls), `perlio` (re-implementation of stdio buffering in a portable manner), `crlf` (does CRLF <= "\n" translation as on Win32, but available on any platform). A `mmap` layer may be available if platform supports it (mostly UNIXes).

Layers to be applied by default may be specified via the ‘open’ pragma.

See `for` for the effects of PerlIO on your architecture name.

- If your platform supports `fork()`, you can use the list form of `open` for pipes. For example:

```
open KID_PS, "-|", "ps", "aux" or die $!;
```

forks the `ps(1)` command (without spawning a shell, as there are more than three arguments to `open()`), and reads its standard output via the `KID_PS` filehandle. See [perlipc](#).

- File handles can be marked as accepting Perl’s internal encoding of Unicode (UTF–8 or UTF–EBCDIC depending on platform) by a pseudo layer “:utf8”:

```
open($fh, ">:utf8", "Uni.txt");
```

Note for EBCDIC users: the pseudo layer “:utf8” is erroneously named for you since it’s not UTF–8 what you will be getting but instead UTF–EBCDIC. See [perlunicode](#), [utf8](#), and <http://www.unicode.org/unicode/reports/tr16/> for more information. In future releases this naming may change. See [perluniintro](#) for more information about UTF–8.

- If your environment variables (`LC_ALL`, `LC_CTYPE`, `LANG`, `LANGUAGE`) look like you want to use UTF–8 (any of the the variables match `/utf-?8/i`), your `STDIN`, `STDOUT`, `STDERR` handles and the default open layer (see [open](#)) are marked as UTF–8. (This feature, like other new features that combine Unicode and I/O, work only if you are using PerlIO, but that’s the default.)

Note that after this Perl really does assume that everything is UTF–8: for example if some input handle is not, Perl will probably very soon complain about the input data like this “Malformed UTF–8 ...” since any old eight–bit data is not legal UTF–8.

Note for code authors: if you want to enable your users to use UTF–8 as their default encoding but in your code still have eight–bit I/O streams (such as images or zip files), you need to explicitly `open()` or `binmode()` with `:bytes` (see [open](#) and [binmode](#)), or you can just use `binmode(FH)` (nice for pre–5.8.0 backward compatibility).

- File handles can translate character encodings from/to Perl’s internal Unicode form on read/write via the “:encoding()” layer.
- File handles can be opened to “in memory” files held in Perl scalars via:

```
open($fh, '>', \$variable) || ...
```

- Anonymous temporary files are available without need to ‘use FileHandle’ or other module via

```
open($fh, "+>", undef) || ...
```

That is a literal `undef`, not an undefined value.

ithreads

The new interpreter threads (“ithreads” for short) implementation of multithreading, by Arthur Bergman, replaces the old “5.005 threads” implementation. In the `ithreads` model any data sharing between threads must be explicit, as opposed to the model where data sharing was implicit. See [threads](#) and [threads::shared](#), and [perlthrtut](#).

As a part of the `ithreads` implementation Perl will also use any necessary and detectable reentrant libc interfaces.

Restricted Hashes

A restricted hash is restricted to a certain set of keys, no keys outside the set can be added. Also individual keys can be restricted so that the key cannot be deleted and the value cannot be changed. No new syntax is involved: the `Hash::Util` module is the interface.

Safe Signals

Perl used to be fragile in that signals arriving at inopportune moments could corrupt Perl's internal state. Now Perl postpones handling of signals until it's safe (between opcodes).

This change may have surprising side effects because signals no longer interrupt Perl instantly. Perl will now first finish whatever it was doing, like finishing an internal operation (like `sort()`) or an external operation (like an I/O operation), and only then look at any arrived signals (and before starting the next operation). No more corrupt internal state since the current operation is always finished first, but the signal may take more time to get heard. Note that breaking out from potentially blocking operations should still work, though.

Understanding of Numbers

In general a lot of fixing has happened in the area of Perl's understanding of numbers, both integer and floating point. Since in many systems the standard number parsing functions like `strtoul()` and `atof()` seem to have bugs, Perl tries to work around their deficiencies. This results hopefully in more accurate numbers.

Perl now tries internally to use integer values in numeric conversions and basic arithmetics (+ - * /) if the arguments are integers, and tries also to keep the results stored internally as integers. This change leads to often slightly faster and always less lossy arithmetics. (Previously Perl always preferred floating point numbers in its math.)

Arrays now always interpolate into double-quoted strings [561]

In double-quoted strings, arrays now interpolate, no matter what. The behavior in earlier versions of perl 5 was that arrays would interpolate into strings if the array had been mentioned before the string was compiled, and otherwise Perl would raise a fatal compile-time error. In versions 5.000 through 5.003, the error was

```
Literal @example now requires backslash
```

In versions 5.004_01 through 5.6.0, the error was

```
In string, @example now must be written as \@example
```

The idea here was to get people into the habit of writing `"fred\@example.com"` when they wanted a literal `@` sign, just as they have always written `"Give me back my \$5"` when they wanted a literal `$` sign.

Starting with 5.6.1, when Perl now sees an `@` sign in a double-quoted string, it *always* attempts to interpolate an array, regardless of whether or not the array has been used or declared already. The fatal error has been downgraded to an optional warning:

```
Possible unintended interpolation of @example in string
```

This warns you that `"fred@example.com"` is going to turn into `fred.com` if you don't backslash the `@`. See <http://www.plover.com/~mjd/perl/at-error.html> for more details about the history here.

Miscellaneous Changes

- AUTOLOAD is now lvaluable, meaning that you can add the `:lvalue` attribute to AUTOLOAD subroutines and you can assign to the AUTOLOAD return value.
- The `$Config{byteorder}` (and corresponding `BYTEORDER` in `config.h`) was previously wrong in platforms if `sizeof(long)` was 4, but `sizeof(IV)` was 8. The `byteorder` was only `sizeof(long)` bytes long (1234 or 4321), but now it is correctly `sizeof(IV)` bytes long, (12345678 or 87654321). (This problem didn't affect Windows platforms.)

Also, `$Config{byteorder}` is now computed dynamically—this is more robust with "fat binaries" where an executable image contains binaries for more than one binary platform, and when cross-compiling.

- `perl -d:Module=arg,arg,arg` now works (previously one couldn't pass in multiple arguments.)
- `do` followed by a bareword now ensures that this bareword isn't a keyword (to avoid a bug where `do q(foo.pl)` tried to call a subroutine called `q`). This means that for example instead of `do format()` you must write `do &format()`.
- The builtin `dump()` now gives an optional warning `dump()` better written as `CORE::dump()`, meaning that by default `dump(...)` is resolved as the builtin `dump()` which dumps core and aborts, not as (possibly) user-defined sub `dump`. To call the latter, qualify the call as `&dump(...)`. (The whole `dump()` feature is to be considered deprecated, and possibly removed/changed in future releases.)
- `chomp()` and `chop()` are now overridable. Note, however, that their prototype (as given by `prototype("CORE::chomp")`) is undefined, because it cannot be expressed and therefore one cannot really write replacements to override these builtins.
- `END` blocks are now run even if you `exit/die` in a `BEGIN` block. Internally, the execution of `END` blocks is now controlled by `PL_exit_flags` & `PERL_EXIT_DESTRUCT_END`. This enables the new behaviour for Perl embedders. This will default in 5.10. See [perlembed](#).
- Formats now support zero-padded decimal fields.
- Although "you shouldn't do that", it was possible to write code that depends on Perl's hashed key order (`Data::Dumper` does this). The new algorithm "One-at-a-Time" produces a different hashed key order. More details are in [perlfaq](#).
- `lstat(FILEHANDLE)` now gives a warning because the operation makes no sense. In future releases this may become a fatal error.
- Spurious syntax errors generated in certain situations, when `glob()` caused `File::Glob` to be loaded for the first time, have been fixed. [561]
- `Lvalue` subroutines can now return `undef` in list context. However, the `lvalue` subroutine feature still remains experimental. [561+]
- A lost warning "Can't declare ... dereference in my" has been restored (Perl had it earlier but it became lost in later releases.)
- A new special regular expression variable has been introduced: `$_N`, which contains the most-recently closed group (submatch).
- `no Module;` does not produce an error even if `Module` does not have an `unimport()` method. This parallels the behavior of `use` vis-à-vis `import`. [561]
- The numerical comparison operators return `undef` if either operand is a NaN. Previously the behaviour was unspecified.
- `our` can now have an experimental optional attribute `unique` that affects how global variables are shared among multiple interpreters, see [our](#).
- The following builtin functions are now overridable: `each()`, `keys()`, `pop()`, `push()`, `shift()`, `splice()`, `unshift()`. [561]
- `pack()` / `unpack()` can now group template letters with `()` and then apply repetition/count modifiers on the groups.
- `pack()` / `unpack()` can now process the Perl internal numeric types: IVs, UVs, NVs— and also long doubles, if supported by the platform. The template letters are `j`, `J`, `F`, and `D`.
- `pack('U0a*', ...)` can now be used to force a string to UTF8.

- `my __PACKAGE__ $obj` now works. [561]
- `POSIX::sleep()` now returns the number of *unslept* seconds (as the POSIX standard says), as opposed to `CORE::sleep()` which returns the number of slept seconds.
- The `printf()` and `sprintf()` now support parameter reordering using the `%\d+\$` and `*\d+\$` syntaxes. For example


```
print "%2\$s %1\$s\n", "foo", "bar";
```

 will print "bar foo\n". This feature helps in writing internationalised software, and in general when the order of the parameters can vary.
- The `(\&)` prototype now works properly. [561]
- `prototype([\$_&])` is now available to implicitly create references (useful for example if you want to emulate the `tie()` interface).
- A new command-line option, `-t` is available. It is the little brother of `-T`: instead of dying on taint violations, lexical warnings are given. **This is only meant as a temporary debugging aid while securing the code of old legacy applications. This is not a substitute for `-T`.**
- In other taint news, the `exec LIST` and `system LIST` have now been considered too risky (think `exec @ARGV`: it can start any program with any arguments), and now the said forms cause a warning under lexical warnings. You should carefully launder the arguments to guarantee their validity. In future releases of Perl the forms will become fatal errors so consider starting laundering now.
- Tied hash interfaces are now required to have the `EXISTS` and `DELETE` methods (either own or inherited).
- If `tr///` is just counting characters, it doesn't attempt to modify its target.
- `untie()` will now call an `UNTIE()` hook if it exists. See [perl tie](#) for details. [561]
- [utime](#) now supports `utime undef, undef, @files` to change the file timestamps to the current time.
- The rules for allowing underscores (underbars) in numeric constants have been relaxed and simplified: now you can have an underscore simply **between digits**.
- Rather than relying on C's `argv[0]` (which may not contain a full pathname) where possible `$^X` is now set by asking the operating system. (eg by reading */proc/self/exe* on Linux, */proc/curproc/file* on FreeBSD)
- A new variable, `$_{^TAINT}`, indicates whether taint mode is enabled.
- You can now override the `readline()` builtin, and this overrides also the `<FILEHANDLE` angle bracket operator.
- The command-line options `-s` and `-F` are now recognized on the shebang (`#!`) line.
- Use of the `/c` match modifier without an accompanying `/g` modifier elicits a new warning: `Use of /c modifier is meaningless without /g`.
 Use of `/c` in substitutions, even with `/g`, elicits `Use of /c modifier is meaningless in s///`.
 Use of `/g` with `split` elicits `Use of /g modifier is meaningless in split`.
- Support for the `CLONE` special subroutine had been added. With `ithreads`, when a new thread is created, all Perl data is cloned, however non-Perl data cannot be cloned automatically. In `CLONE` you can do whatever you need to do, like for example handle the cloning of non-Perl data, if necessary. `CLONE` will be executed once for every package that has it defined or inherited. It will be called in the context of the new thread, so all modifications are made in the new area.

See [perlmod](#)

Modules and Pragmata

New Modules and Pragmata

- `Attribute::Handlers`, originally by Damian Conway and now maintained by Arthur Bergman, allows a class to define attribute handlers.

```
package MyPack;
use Attribute::Handlers;
sub Wolf :ATTR(SCALAR) { print "howl!\n" }

# later, in some package using or inheriting from MyPack...

my MyPack $Fluffy : Wolf; # the attribute handler Wolf will be called
```

Both variables and routines can have attribute handlers. Handlers can be specific to type (SCALAR, ARRAY, HASH, or CODE), or specific to the exact compilation phase (BEGIN, CHECK, INIT, or END). See [Attribute::Handlers](#).

- `B::Concise`, by Stephen McCamant, is a new compiler backend for walking the Perl syntax tree, printing concise info about ops. The output is highly customisable. See [B::Concise](#). [561+]
- The new `bignum`, `bigint`, and `bigrat` pragmas, by Tels, implement transparent bignum support (using the `Math::BigInt`, `Math::BigFloat`, and `Math::BigRat` backends).
- `Class::ISA`, by Sean Burke, is a module for reporting the search path for a class's ISA tree. See [Class::ISA](#).
- `Cwd` now has a split personality: if possible, an XS extension is used, (this will hopefully be faster, more secure, and more robust) but if not possible, the familiar Perl implementation is used.
- `Devel::PPPort`, originally by Kenneth Albanowski and now maintained by Paul Marquess, has been added. It is primarily used by `h2xs` to enhance portability of XS modules between different versions of Perl. See [Devel::PPPort](#).
- `Digest`, frontend module for calculating digests (checksums), from Gisle Aas, has been added. See [Digest](#).
- `Digest::MD5` for calculating MD5 digests (checksums) as defined in RFC 1321, from Gisle Aas, has been added. See [Digest::MD5](#).

```
use Digest::MD5 'md5_hex';

$digest = md5_hex("Thirsty Camel");

print $digest, "\n"; # 01d19d9d2045e005c3f1b80e8b164de1
```

NOTE: the MD5 backward compatibility module is deliberately not included since its further use is discouraged.

See also [PerlIO::via::QuotedPrint](#).

- `Encode`, originally by Nick Ing-Simmons and now maintained by Dan Kogai, provides a mechanism to translate between different character encodings. Support for Unicode, ISO-8859-1, and ASCII are compiled in to the module. Several other encodings (like the rest of the ISO-8859, CP*/Win*, Mac, KOI8-R, three variants EBCDIC, Chinese, Japanese, and Korean encodings) are included and can be loaded at runtime. (For space considerations, the largest Chinese encodings have been separated into their own CPAN module, `Encode::HanExtra`, which `Encode` will use if available). See [Encode](#).

Any encoding supported by `Encode` module is also available to the `":encoding()"` layer if `PerlIO` is used.

- `Hash::Util` is the interface to the new *restricted hashes* feature. (Implemented by Jeffrey Friedl, Nick Ing-Simmons, and Michael Schwern.) See [Hash::Util](#).
- `I18N::Langinfo` can be used to query locale information. See [I18N::Langinfo](#).
- `I18N::LangTags`, by Sean Burke, has functions for dealing with RFC3066-style language tags. See [I18N::LangTags](#).
- `ExtUtils::Constant`, by Nicholas Clark, is a new tool for extension writers for generating XS code to import C header constants. See [ExtUtils::Constant](#).
- `Filter::Simple`, by Damian Conway, is an easy-to-use frontend to `Filter::Util::Call`. See [Filter::Simple](#).

```
# in MyFilter.pm:
package MyFilter;

use Filter::Simple sub {
    while (my ($from, $to) = splice @_, 0, 2) {
        s/$from/$to/g;
    }
};

1;

# in user's code:

use MyFilter qr/red/ => 'green';

print "red\n";    # this code is filtered, will print "green\n"
print "bored\n"; # this code is filtered, will print "bogreen\n"

no MyFilter;

print "red\n";    # this code is not filtered, will print "red\n"
```

- `File::Temp`, by Tim Jenness, allows one to create temporary files and directories in an easy, portable, and secure way. See [File::Temp](#). [561+]
- `Filter::Util::Call`, by Paul Marquess, provides you with the framework to write *source filters* in Perl. For most uses, the frontend `Filter::Simple` is to be preferred. See [Filter::Util::Call](#).
- `if`, by Ilya Zakharevich, is a new pragma for conditional inclusion of modules.
- [libnet](#), by Graham Barr, is a collection of perl5 modules related to network programming. See [Net::FTP](#), [Net::NNTP](#), [Net::Ping](#) (not part of libnet, but related), [Net::POP3](#), [Net::SMTP](#), and [Net::Time](#).

Perl installation leaves libnet unconfigured; use [libnetcfg](#) to configure it.

- `List::Util`, by Graham Barr, is a selection of general-utility list subroutines, such as `sum()`, `min()`, `first()`, and `shuffle()`. See [List::Util](#).
- `Locale::Constants`, `Locale::Country`, `Locale::Currency`, `Locale::Language`, and [Locale::Script](#), by Neil Bowers, have been added. They provide the codes for various locale standards, such as "fr" for France, "usd" for US Dollar, and "ja" for Japanese.

```
use Locale::Country;

$country = code2country('jp');           # $country gets 'Japan'
$code    = country2code('Norway');       # $code gets 'no'
```

See [Locale::Constants](#), [Locale::Country](#), [Locale::Currency](#), and [Locale::Language](#).

- `Locale::Maketext`, by Sean Burke, is a localization framework. See [Locale::Maketext](#), and [Locale::Maketext::TPJ13](#). The latter is an article about software localization, originally published in The Perl Journal #13, and republished here with kind permission.
- `Math::BigRat` for big rational numbers, to accompany `Math::BigInt` and `Math::BigFloat`, from Tels. See [Math::BigRat](#).
- `Memoize` can make your functions faster by trading space for time, from Mark-Jason Dominus. See [Memoize](#).
- `MIME::Base64`, by Gisle Aas, allows you to encode data in base64, as defined in RFC 2045 – *MIME (Multipurpose Internet Mail Extensions)*.

```
use MIME::Base64;

$encoded = encode_base64('Aladdin:open sesame');
$decoded = decode_base64($encoded);

print $encoded, "\n"; # "QWxhZGRpbjpvcGVuIHNlc2FtZQ=="
```

See [MIME::Base64](#).

- `MIME::QuotedPrint`, by Gisle Aas, allows you to encode data in quoted-printable encoding, as defined in RFC 2045 – *MIME (Multipurpose Internet Mail Extensions)*.

```
use MIME::QuotedPrint;

$encoded = encode_qp("Smiley in Unicode: \x{263a}");
$decoded = decode_qp($encoded);

print $encoded, "\n"; # "Smiley in Unicode: =263A"
```

See also [PerlIO::via::QuotedPrint](#).

- `NEXT`, by Damian Conway, is a pseudo-class for method redispatch. See [NEXT](#).
- `open` is a new pragma for setting the default I/O layers for `open()`.
- `PerlIO::scalar`, by Nick Ing-Simmons, provides the implementation of IO to "in memory" Perl scalars as discussed above. It also serves as an example of a loadable PerlIO layer. Other future possibilities include `PerlIO::Array` and `PerlIO::Code`. See [PerlIO::scalar](#).
- `PerlIO::via`, by Nick Ing-Simmons, acts as a PerlIO layer and wraps PerlIO layer functionality provided by a class (typically implemented in Perl code).
- `PerlIO::via::QuotedPrint`, by Elizabeth Mattijsen, is an example of a `PerlIO::via` class:

```
use PerlIO::via::QuotedPrint;
open($fh, ">:via(QuotedPrint)", $path);
```

This will automatically convert everything output to `$fh` to Quoted-Printable. See [PerlIO::via](#) and [PerlIO::via::QuotedPrint](#).

- `Pod::ParseLink`, by Russ Allbery, has been added, to parse links in pods as described in the new `perlpodspec`.
- `Pod::Text::Overstrike`, by Joe Smith, has been added. It converts POD data to formatted overstrike text. See [Pod::Text::Overstrike](#). [561+]
- `Scalar::Util` is a selection of general-utility scalar subroutines, such as `blessed()`, `reftype()`, and `tainted()`. See [Scalar::Util](#).
- `sort` is a new pragma for controlling the behaviour of `sort()`.

- Storable gives persistence to Perl data structures by allowing the storage and retrieval of Perl data to and from files in a fast and compact binary format. Because in effect Storable does serialisation of Perl data structures, with it you can also clone deep, hierarchical datastructures. Storable was originally created by Raphael Manfredi, but it is now maintained by Abhijit Menon-Sen. Storable has been enhanced to understand the two new hash features, Unicode keys and restricted hashes. See [Storable](#).
- Switch, by Damian Conway, has been added. Just by saying

```
use Switch;
```

you have switch and case available in Perl.

```
use Switch;

switch ($val) {

    case 1          { print "number 1" }
    case "a"        { print "string a" }
    case [1..10,42] { print "number in list" }
    case (@array)   { print "number in list" }
    case /\w+/      { print "pattern" }
    case qr/\w+/    { print "pattern" }
    case (%hash)    { print "entry in hash" }
    case (\%hash)   { print "entry in hash" }
    case (\&sub)    { print "arg to subroutine" }
    else           { print "previous case not true" }

}
```

See [Switch](#).

- Test::More, by Michael Schwern, is yet another framework for writing test scripts, more extensive than Test::Simple. See [Test::More](#).
- Test::Simple, by Michael Schwern, has basic utilities for writing tests. See [Test::Simple](#).
- Text::Balanced, by Damian Conway, has been added, for extracting delimited text sequences from strings.

```
use Text::Balanced 'extract_delimited';
```

```
($a, $b) = extract_delimited("'never say never', he never said", "'", '');
```

\$a will be "'never say never'", \$b will be ', he never said'.

In addition to `extract_delimited()`, there are also `extract_bracketed()`, `extract_quotelike()`, `extract_codeblock()`, `extract_variable()`, `extract_tagged()`, `extract_multiple()`, `gen_delimited_pat()`, and `gen_extract_tagged()`. With these, you can implement rather advanced parsing algorithms. See [Text::Balanced](#).

- threads, by Arthur Bergman, is an interface to interpreter threads. Interpreter threads (ithreads) is the new thread model introduced in Perl 5.6 but only available as an internal interface for extension writers (and for Win32 Perl for `fork()` emulation). See [threads](#), [threads::shared](#), and [perlthrtut](#).
- threads::shared, by Arthur Bergman, allows data sharing for interpreter threads. See [threads::shared](#).
- Tie::File, by Mark-Jason Dominus, associates a Perl array with the lines of a file. See [Tie::File](#).
- Tie::Memoize, by Ilya Zakharevich, provides on-demand loaded hashes. See [Tie::Memoize](#).

- `Tie::RefHash::Nestable`, by Edward Avis, allows storing hash references (unlike the standard `Tie::RefHash`) The module is contained within `Tie::RefHash`. See [Tie::RefHash](#).
- `Time::HiRes`, by Douglas E. Wegscheid, provides high resolution timing (`ualarm`, `usleep`, and `gettimeofday`). See [Time::HiRes](#).
- `Unicode::UCD` offers a querying interface to the Unicode Character Database. See [Unicode::UCD](#).
- `Unicode::Collate`, by SADAHIRO Tomoyuki, implements the UCA (Unicode Collation Algorithm) for sorting Unicode strings. See [Unicode::Collate](#).
- `Unicode::Normalize`, by SADAHIRO Tomoyuki, implements the various Unicode normalization forms. See [Unicode::Normalize](#).
- `XS::APITest`, by Tim Jenness, is a test extension that exercises XS APIs. Currently only `printf()` is tested: how to output various basic data types from XS.
- `XS::Typemap`, by Tim Jenness, is a test extension that exercises XS typemaps. Nothing gets installed, but the code is worth studying for extension writers.

Updated And Improved Modules and Pragmata

- The following independently supported modules have been updated to the newest versions from CPAN: `CGI`, `CPAN`, `DB_File`, `File::Spec`, `File::Temp`, `Getopt::Long`, `Math::BigFloat`, `Math::BigInt`, the podlators bundle (`Pod::Man`, `Pod::Text`), `Pod::LaTeX` [561+], `Pod::Parser`, `Storable`, `Term::ANSIColor`, `Test`, `Text-Tabs+Wrap`.
- `attributes::reftype()` now works on tied arguments.
- `AutoLoader` can now be disabled with `no AutoLoader;`.
- `B::Deparse` has been significantly enhanced by Robin Houston. It can now deparse almost all of the standard test suite (so that the tests still succeed). There is a make target "test.deparse" for trying this out.
- `Carp` now has better interface documentation, and the `@CARP_NOT` interface has been added to get optional control over where errors are reported independently of `@ISA`, by Ben Tilly.
- `Class::Struct` can now define the classes in compile time.
- `Class::Struct` now assigns the array/hash element if the accessor is called with an array/hash element as the **sole** argument.
- The return value of `Cwd::fastcwd()` is now tainted.
- `Data::Dumper` now has an option to sort hashes.
- `Data::Dumper` now has an option to dump code references using `B::Deparse`.
- `DB_File` now supports newer Berkeley DB versions, among other improvements.
- `Devel::Peek` now has an interface for the Perl memory statistics (this works only if you are using perl's malloc, and if you have compiled with debugging).
- The `English` module can now be used without the infamous performance hit by saying

```
use English '-no_match_vars';
```

(Assuming, of course, that you don't need the troublesome variables `$'`, `$&`, or `$'.`) Also, introduced `@LAST_MATCH_START` and `@LAST_MATCH_END` English aliases for `@-` and `@+`.

- `ExtUtils::MakeMaker` has been significantly cleaned up and fixed. The enhanced version has also been backported to earlier releases of Perl and submitted to CPAN so that the earlier releases can enjoy the fixes.

- The arguments of `WriteMakefile()` in `Makefile.PL` are now checked for sanity much more carefully than before. This may cause new warnings when modules are being installed. See [ExtUtils::MakeMaker](#) for more details.
- `ExtUtils::MakeMaker` now uses `File::Spec` internally, which hopefully leads to better portability.
- `Fcntl`, `Socket`, and `Sys::Syslog` have been rewritten by Nicholas Clark to use the new-style constant dispatch section (see [ExtUtils::Constant](#)). This means that they will be more robust and hopefully faster.
- `File::Find` now `chdir()`s correctly when chasing symbolic links. [561]
- `File::Find` now has pre- and post-processing callbacks. It also correctly changes directories when chasing symbolic links. Callbacks (naughtily) exiting with "next;" instead of "return;" now work.
- `File::Find` is now (again) reentrant. It also has been made more portable.
- The warnings issued by `File::Find` now belong to their own category. You can enable/disable them with `use/no warnings 'File::Find';`.
- `File::Glob::glob()` has been renamed to `File::Glob::bsd_glob()` because the name clashes with the builtin `glob()`. The older name is still available for compatibility, but is deprecated. [561]
- `File::Glob` now supports `GLOB_LIMIT` constant to limit the size of the returned list of filenames.
- `IPC::Open3` now allows the use of numeric file descriptors.
- `IO::Socket` now has an `atmark()` method, which returns true if the socket is positioned at the out-of-band mark. The method is also exportable as a `socketatmark()` function.
- `IO::Socket::INET` failed to open the specified port if the service name was not known. It now correctly uses the supplied port number as is. [561]
- `IO::Socket::INET` has support for the `ReusePort` option (if your platform supports it). The `Reuse` option now has an alias, `ReuseAddr`. For clarity, you may want to prefer `ReuseAddr`.
- `IO::Socket::INET` now supports a value of zero for `LocalPort` (usually meaning that the operating system will make one up.)
- 'use lib' now works identically to @INC. Removing directories with 'no lib' now works.
- `Math::BigFloat` and `Math::BigInt` have undergone a full rewrite by Tels. They are now magnitudes faster, and they support various bignum libraries such as GMP and PARI as their backends.
- `Math::Complex` handles inf, NaN etc., better.
- `Net::Ping` has been considerably enhanced by Rob Brown: multihoming is now supported, Win32 functionality is better, there is now time measuring functionality (optionally high-resolution using `Time::HiRes`), and there is now "external" protocol which uses `Net::Ping::External` module which runs your external ping utility and parses the output. A version of `Net::Ping::External` is available in CPAN.

Note that some of the `Net::Ping` tests are disabled when running under the Perl distribution since one cannot assume one or more of the following: enabled echo port at localhost, full Internet connectivity, or sympathetic firewalls. You can set the environment variable `PERL_TEST_Net_Ping` to "1" (one) before running the Perl test suite to enable all the `Net::Ping` tests.

- `POSIX::sigaction()` is now much more flexible and robust. You can now install coderef handlers, 'DEFAULT', and 'IGNORE' handlers, installing new handlers was not atomic.
- In Safe, `%INC` is now localised in a Safe compartment so that `use/require` work.

- In SDBM_File on dosish platforms, some keys went missing because of lack of support for files with "holes". A workaround for the problem has been added.
- In Search::Dict one can now have a pre-processing hook for the lines being searched.
- The Shell module now has an OO interface.
- In Sys::Syslog there is now a failover mechanism that will go through alternative connection mechanisms until the message is successfully logged.
- The Test module has been significantly enhanced.
- Time::Local::timelocal() does not handle fractional seconds anymore. The rationale is that neither does localtime(), and timelocal() and localtime() are supposed to be inverses of each other.
- The vars pragma now supports declaring fully qualified variables. (Something that our() does not and will not support.)
- The utf8::name space (as in the pragma) provides various Perl-callable functions to provide low level access to Perl's internal Unicode representation. At the moment only length() has been implemented.

Utility Changes

- Emacs perl mode (emacs/cperl-mode.el) has been updated to version 4.31.
- *emacs/e2ctags.pl* is now much faster.
- enc2xs is a tool for people adding their own encodings to the Encode module.
- h2ph now supports C trigraphs.
- h2xs now produces a template README.
- h2xs now uses Devel::PPPort for better portability between different versions of Perl.
- h2xs uses the new *ExtUtils::Constant/ExtUtils::Constant* module which will affect newly created extensions that define constants. Since the new code is more correct (if you have two constants where the first one is a prefix of the second one, the first constant **never** got defined), less lossy (it uses integers for integer constant, as opposed to the old code that used floating point numbers even for integer constants), and slightly faster, you might want to consider regenerating your extension code (the new scheme makes regenerating easy). *h2xs* now also supports C trigraphs.
- libnetcfg has been added to configure libnet.
- perlbug is now much more robust. It also sends the bug report to perl.org, not perl.com.
- perlcc has been rewritten and its user interface (that is, command line) is much more like that of the UNIX C compiler, cc. (The perlbc tools has been removed. Use perlcc -B instead.) **Note that perlcc is still considered very experimental and unsupported.** [561]
- perlivp is a new Installation Verification Procedure utility for running any time after installing Perl.
- piconv is an implementation of the character conversion utility iconv, demonstrating the new Encode module.
- pod2html now allows specifying a cache directory.
- pod2html now produces XHTML 1.0.
- pod2html now understands POD written using different line endings (PC-like CRLF versus UNIX-like LF versus MacClassic-like CR).

- `s2p` has been completely rewritten in Perl. (It is in fact a full implementation of `sed` in Perl: you can use the `sed` functionality by using the `psed` utility.)
- `xsubpp` now understands POD documentation embedded in the `*.xs` files. [561]
- `xsubpp` now supports the `OUT` keyword.

New Documentation

- `perl56delta` details the changes between the 5.005 release and the 5.6.0 release.
- `perlclib` documents the internal replacements for standard C library functions. (Interesting only for extension writers and Perl core hackers.) [561+]
- `perldebtut` is a Perl debugging tutorial. [561+]
- `perlebcdic` contains considerations for running Perl on EBCDIC platforms. [561+]
- `perlintro` is a gentle introduction to Perl.
- `perliol` documents the internals of PerlIO with layers.
- `perlmodstyle` is a style guide for writing modules.
- `perlnewmod` tells about writing and submitting a new module. [561+]
- `perlpacktut` is a `pack()` tutorial.
- `perlpod` has been rewritten to be clearer and to record the best practices gathered over the years.
- `perlpodspec` is a more formal specification of the pod format, mainly of interest for writers of pod applications, not to people writing in pod.
- `perlretut` is a regular expression tutorial. [561+]
- `perlrequick` is a regular expressions quick-start guide. Yes, much quicker than `perlretut`. [561]
- `perltodo` has been updated.
- `perltotc` has been renamed as `perltoc` (to not to conflict with `perltot` in filesystems restricted to "8.3" names).
- `perluniintro` is an introduction to using Unicode in Perl. (`perlunicode` is more of a detailed reference and background information)
- `perlutil` explains the command line utilities packaged with the Perl distribution. [561+]

The following platform-specific documents are available before the installation as `README.platform`, and after the installation as `perlplatform`:

```
perlaix perlamiga perlapollo perlbeos perlbs2000
perlce perlcygwin perldgux perldos perlepoc perlfreesbsd perlhpux
perlhurd perlirix perlmachten perlmacos perlminit perlmpaix
perlntware perlps2 perlps390 perlplan9 perlqnx perlsolaris
perltru64 perluts perlvmesa perlvm perlvs perlwin32
```

These documents usually detail one or more of the following subjects: configuring, building, testing, installing, and sometimes also using Perl on the said platform.

Eastern Asian Perl users are now welcomed in their own languages: `README.jp` (Japanese), `README.ko` (Korean), `README.cn` (simplified Chinese) and `README.tw` (traditional Chinese), which are written in normal pod but encoded in EUC-JP, EUC-KR, EUC-CN and Big5. These will get installed as

```
perljp perlko perlcn perltw
```

- The documentation for the POSIX–BC platform is called "BS2000", to avoid confusion with the Perl POSIX module.
- The documentation for the WinCE platform is called perlce (README.ce in the source code kit), to avoid confusion with the perlwin32 documentation on 8.3–restricted filesystems.

Performance Enhancements

- `map()` could get pathologically slow when the result list it generates is larger than the source list. The performance has been improved for common scenarios. [561]
- `sort()` is also fully reentrant, in the sense that the sort function can itself call `sort()`. This did not work reliably in previous releases. [561]
- `sort()` has been changed to use primarily mergesort internally as opposed to the earlier quicksort. For very small lists this may result in slightly slower sorting times, but in general the speedup should be at least 20%. Additional bonuses are that the worst case behaviour of `sort()` is now better (in computer science terms it now runs in time $O(N \log N)$, as opposed to quicksort's $\Theta(N^2)$ worst–case run time behaviour), and that `sort()` is now stable (meaning that elements with identical keys will stay ordered as they were before the sort). See the `sort` pragma for information.

The story in more detail: suppose you want to serve yourself a little slice of Pi.

```
@digits = ( 3,1,4,1,5,9 );
```

A numerical sort of the digits will yield (1,1,3,4,5,9), as expected. Which 1 comes first is hard to know, since one 1 looks pretty much like any other. You can regard this as totally trivial, or somewhat profound. However, if you just want to sort the even digits ahead of the odd ones, then what will

```
sort { ($a % 2) <=> ($b % 2) } @digits;
```

yield? The only even digit, 4, will come first. But how about the odd numbers, which all compare equal? With the quicksort algorithm used to implement Perl 5.6 and earlier, the order of ties is left up to the sort. So, as you add more and more digits of Pi, the order in which the sorted even and odd digits appear will change. and, for sufficiently large slices of Pi, the quicksort algorithm in Perl 5.8 won't return the same results even if reinvoked with the same input. The justification for this rests with quicksort's worst case behavior. If you run

```
sort { $a <=> $b } ( 1 .. $N , 1 .. $N );
```

(something you might approximate if you wanted to merge two sorted arrays using `sort`), doubling `$N` doesn't just double the quicksort time, it *quadruples* it. Quicksort has a worst case run time that can grow like N^2 , so–called *quadratic* behaviour, and it can happen on patterns that may well arise in normal use. You won't notice this for small arrays, but you *will* notice it with larger arrays, and you may not live long enough for the sort to complete on arrays of a million elements. So the 5.8 quicksort scrambles large arrays before sorting them, as a statistical defence against quadratic behaviour. But that means if you sort the same large array twice, ties may be broken in different ways.

Because of the unpredictability of tie–breaking order, and the quadratic worst–case behaviour, quicksort was *almost* replaced completely with a stable mergesort. *Stable* means that ties are broken to preserve the original order of appearance in the input array. So

```
sort { ($a % 2) <=> ($b % 2) } (3,1,4,1,5,9);
```

will yield (4,3,1,1,5,9), guaranteed. The even and odd numbers appear in the output in the same order they appeared in the input. Mergesort has worst case $O(N \log N)$ behaviour, the best value attainable. And, ironically, this mergesort does particularly well where quicksort goes quadratic: mergesort sorts $(1..N, 1..N)$ in $O(N)$ time. But quicksort was rescued at the last moment because it is faster than mergesort on certain inputs and platforms. For example, if you really *don't* care about the order of even and odd digits, quicksort will run in $O(N)$ time; it's very good at sorting many repetitions of a small number of distinct elements. The quicksort divide and conquer strategy works well on platforms

with relatively small, very fast, caches. Eventually, the problem gets whittled down to one that fits in the cache, from which point it benefits from the increased memory speed.

Quicksort was rescued by implementing a sort pragma to control aspects of the sort. The **stable** subpragma forces stable behaviour, regardless of algorithm. The **_quicksort** and **_mergesort** subpragmas are heavy-handed ways to select the underlying implementation. The leading **_** is a reminder that these subpragmas may not survive beyond 5.8. More appropriate mechanisms for selecting the implementation exist, but they wouldn't have arrived in time to save quicksort.

- Hashes now use Bob Jenkins "One-at-a-Time" hashing key algorithm (<http://burtleburtle.net/bob/hash/doobs.html>). This algorithm is reasonably fast while producing a much better spread of values than the old hashing algorithm (originally by Chris Torek, later tweaked by Ilya Zakharevich). Hash values output from the algorithm on a hash of all 3-char printable ASCII keys comes much closer to passing the DIEHARD random number generation tests. According to perlbench, this change has not affected the overall speed of Perl.
- `unshift()` should now be noticeably faster.

Installation and Configuration Improvements

Generic Improvements

- **INSTALL** now explains how you can configure Perl to use 64-bit integers even on non-64-bit platforms.
- **Policy.sh** policy change: if you are reusing a **Policy.sh** file (see **INSTALL**) and you use **Configure -Dprefix=/foo/bar** and in the old **Policy** `$prefix eq $siteprefix` and `$prefix eq $vendorprefix`, all of them will now be changed to the new prefix, `/foo/bar`. (Previously only `$prefix` changed.) If you do not like this new behaviour, specify `prefix`, `siteprefix`, and `vendorprefix` explicitly.
- A new optional location for Perl libraries, `otherlibdirs`, is available. It can be used for example for vendor add-ons without disturbing Perl's own library directories.
- In many platforms, the vendor-supplied 'cc' is too stripped-down to build Perl (basically, 'cc' doesn't do ANSI C). If this seems to be the case and 'cc' does not seem to be the GNU C compiler 'gcc', an automatic attempt is made to find and use 'gcc' instead.
- gcc needs to closely track the operating system release to avoid build problems. If **Configure** finds that gcc was built for a different operating system release than is running, it now gives a clearly visible warning that there may be trouble ahead.
- Since Perl 5.8 is not binary-compatible with previous releases of Perl, **Configure** no longer suggests including the 5.005 modules in `@INC`.
- **Configure -S** can now run non-interactively. [561]
- **Configure** support for pdp11-style memory models has been removed due to obsolescence. [561]
- **configure.gnu** now works with options with whitespace in them.
- **installperl** now outputs everything to **STDERR**.
- Because **PerlIO** is now the default on most platforms, `"-perlio"` doesn't get appended to the `$Config{archname}` (also known as `$^O`) anymore. Instead, if you explicitly choose not to use **perlio** (**Configure** command line option `-Uuseperlio`), you will get `"-stdio"` appended.
- Another change related to the architecture name is that `"-64all"` (`-Duse64bitall`, or "maximally 64-bit") is appended only if your pointers are 64 bits wide. (To be exact, the `use64bitall` is ignored.)
- In AFS installations, one can configure the root of the AFS to be somewhere else than the default `/afs` by using the **Configure** parameter `-Dafsroot=/some/where/else`.

- APPLLIB_EXP, a lesser-known configuration-time definition, has been documented. It can be used to prepend site-specific directories to Perl's default search path (@INC); see INSTALL for information.
- The version of Berkeley DB used when the Perl (and, presumably, the DB_File extension) was built is now available as @Config{qw(db_version_major db_version_minor db_version_patch)} from Perl and as DB_VERSION_MAJOR_CFG DB_VERSION_MINOR_CFG DB_VERSION_PATCH_CFG from C.
- Building Berkeley DB3 for compatibility modes for DB, NDBM, and ODBM has been documented in INSTALL.
- If you have CPAN access (either network or a local copy such as a CD-ROM) you can during specify extra modules to Configure to build and install with Perl using the `-Dextras=...` option. See INSTALL for more details.
- In addition to `config.over`, a new override file, `config.arch`, is available. This file is supposed to be used by hints file writers for architecture-wide changes (as opposed to `config.over` which is for site-wide changes).
- If your file system supports symbolic links, you can build Perl outside of the source directory by

```
mkdir /tmp/perl/build/directory
cd /tmp/perl/build/directory
sh /path/to/perl/source/Configure -Dmk symlinks ...
```

This will create in `/tmp/perl/build/directory` a tree of symbolic links pointing to files in `/path/to/perl/source`. The original files are left unaffected. After Configure has finished, you can just say

```
make all test
```

and Perl will be built and tested, all in `/tmp/perl/build/directory`. [561]

- For Perl developers, several new make targets for profiling and debugging have been added; see [perlhack](#).
 - Use of the *gprof* tool to profile Perl has been documented in [perlhack](#). There is a make target called "perl.gprof" for generating a gprofiled Perl executable.
 - If you have GCC 3, there is a make target called "perl.gcov" for creating a gcov'd Perl executable for coverage analysis. See [perlhack](#).
 - If you are on IRIX or Tru64 platforms, new profiling/debugging options have been added; see [perlhack](#) for more information about pixie and Third Degree.
- Guidelines of how to construct minimal Perl installations have been added to INSTALL.
- The Thread extension is now not built at all under `ithreads` (`Configure -Duseithreads`) because it wouldn't work anyway (the Thread extension requires being Configured with `-Duse5005threads`).

Note that the 5.005 threads are unsupported and deprecated: if you have code written for the old threads you should migrate it to the new ithreads model.

- The Gconvert macro (`$Config{d_Gconvert}`) used by perl for stringifying floating-point numbers is now more picky about using `printf %.*g` rules for the conversion. Some platforms that used to use `gcvt` may now resort to the slower `sprintf`.
- The obsolete method of making a special (e.g., debugging) flavor of perl by saying

```
make LIBPERL=libperl.d.a
```

has been removed. Use `-DDEBUGGING` instead.

New Or Improved Platforms

For the list of platforms known to support Perl, see [Supported Platforms in perlport](#).

- AIX dynamic loading should be now better supported.
- AIX should now work better with gcc, threads, and 64-bitness. Also the long doubles support in AIX should be better now. See [perlaix](#).
- AtheOS (<http://www.atheos.cx/>) is a new platform.
- BeOS has been reclaimed.
- The DG/UX platform now supports 5.005-style threads. See [perldgux](#).
- The DYNIX/ptx platform (also known as dynixptx) is supported at or near osvers 4.5.2.
- EBCDIC platforms (z/OS (also known as OS/390), POSIX-BC, and VM/ESA) have been regained. Many test suite tests still fail and the co-existence of Unicode and EBCDIC isn't quite settled, but the situation is much better than with Perl 5.6. See [perlos390](#), [perlbs2000](#) (for POSIX-BC), and [perlvmesa](#) for more information.
- Building perl with `-Duseithreads` or `-Duse5005threads` now works under HP-UX 10.20 (previously it only worked under 10.30 or later). You will need a thread library package installed. See [README.hpux](#). [561]
- Mac OS Classic is now supported in the mainstream source package (MacPerl has of course been available since perl 5.004 but now the source code bases of standard Perl and MacPerl have been synchronised) [561]
- Mac OS X (or Darwin) should now be able to build Perl even on HFS+ filesystems. (The case-insensitivity used to confuse the Perl build process.)
- NCR MP-RAS is now supported. [561]
- All the NetBSD specific patches (except for the installation specific ones) have been merged back to the main distribution.
- NetWare from Novell is now supported. See [perlnetware](#).
- NonStop-UX is now supported. [561]
- NEC SUPER-UX is now supported.
- All the OpenBSD specific patches (except for the installation specific ones) have been merged back to the main distribution.
- Perl has been tested with the GNU pth userlevel thread package (<http://www.gnu.org/software/pth/pth.html>). All thread tests of Perl now work, but not without adding some `yield()`s to the tests, so while pth (and other userlevel thread implementations) can be considered to be "working" with Perl ithreads, keep in mind the possible non-preemptability of the underlying thread implementation.
- Stratus VOS is now supported using Perl's native build method (Configure). This is the recommended method to build Perl on VOS. The older methods, which build miniperl, are still available. See [perlvos](#). [561+]
- The Amdahl UTS UNIX mainframe platform is now supported. [561]
- WinCE is now supported. See [perlce](#).
- z/OS (formerly known as OS/390, formerly known as MVS OE) now has support for dynamic loading. This is not selected by default, however, you must specify `-Dusedl` in the arguments of Configure. [561]

Selected Bug Fixes

Numerous memory leaks and uninitialized memory accesses have been hunted down. Most importantly, anonymous subs used to leak quite a bit. [561]

- The `autouse` pragma didn't work for `Multi::Part::Function::Names`.
- `caller()` could cause core dumps in certain situations. `Carp` was sometimes affected by this problem. In particular, `caller()` now returns a subroutine name of `(unknown)` for subroutines that have been removed from the symbol table.
- `chop(@list)` in list context returned the characters chopped in reverse order. This has been reversed to be in the right order. [561]
- `Configure` no longer includes the DBM libraries (`dbm`, `gdbm`, `db`, `ndbm`) when building the Perl binary. The only exception to this is SunOS 4.x, which needs them. [561]
- The behaviour of non-decimal but numeric string constants such as `"0x23"` was platform-dependent: in some platforms that was seen as 35, in some as 0, in some as a floating point number (don't ask). This was caused by Perl's using the operating system libraries in a situation where the result of the string to number conversion is undefined: now Perl consistently handles such strings as zero in numeric contexts.
- Several debugger fixes: exit code now reflects the script exit code, condition `"0"` now treated correctly, the `d` command now checks line number, `$.` no longer gets corrupted, and all debugger output now goes correctly to the socket if `RemotePort` is set. [561]
- The debugger (`perl5db.pl`) has been modified to present a more consistent commands interface, via (`CommandSet=580`). `perl5db.t` was also added to test the changes, and as a placeholder for further tests.

See [perldebug](#).

- The debugger has a new `dumpDepth` option to control the maximum depth to which nested structures are dumped. The `x` command has been extended so that `x N EXPR` dumps out the value of `EXPR` to a depth of at most `N` levels.
- The debugger can now show lexical variables if you have the CPAN module `PadWalker` installed.
- The order of `DESTROY`s has been made more predictable.
- Perl 5.6.0 could emit spurious warnings about redefinition of `dl_error()` when statically building extensions into perl. This has been corrected. [561]
- [dprofpp](#) `-R` didn't work.
- `*foo{FORMAT}` now works.
- Infinity is now recognized as a number.
- `UNIVERSAL::isa` no longer caches methods incorrectly. (This broke the Tk extension with 5.6.0.) [561]
- Lexicals I: lexicals outside an `eval ""` weren't resolved correctly inside a subroutine definition inside the `eval ""` if they were not already referenced in the top level of the `eval""`ed code.
- Lexicals II: lexicals leaked at file scope into subroutines that were declared before the lexicals.
- Lexical warnings now propagating correctly between scopes and into `eval "..."`.
- `use warnings qw(FATAL all)` did not work as intended. This has been corrected. [561]
- `warnings::enabled()` now reports the state of `$^W` correctly if the caller isn't using lexical warnings. [561]

- Line renumbering with `eval` and `#line` now works. [561]
- Fixed numerous memory leaks, especially in `eval ""`.
- Localised tied variables no longer leak memory

```
use Tie::Hash;
tie my %tied_hash => 'Tie::StdHash';

...

# Used to leak memory every time local() was called;
# in a loop, this added up.
local($tied_hash{Foo}) = 1;
```

- Localised hash elements (and `%ENV`) are correctly unlocalised to not exist, if they didn't before they were localised.

```
use Tie::Hash;
tie my %tied_hash => 'Tie::StdHash';

...

# Nothing has set the FOO element so far
{ local $tied_hash{FOO} = 'Bar' }

# This used to print, but not now.
print "exists!\n" if exists $tied_hash{FOO};
```

As a side effect of this fix, tied hash interfaces **must** define the `EXISTS` and `DELETE` methods.

- `mkdir()` now ignores trailing slashes in the directory name, as mandated by POSIX.
- Some versions of glibc have a broken `modfl()`. This affects builds with `-Duselongdouble`. This version of Perl detects this brokenness and has a workaround for it. The glibc release 2.2.2 is known to have fixed the `modfl()` bug.
- Modulus of unsigned numbers now works (`4063328477 % 65535` used to return 27406, instead of 27047). [561]
- Some "not a number" warnings introduced in 5.6.0 eliminated to be more compatible with 5.005. Infinity is now recognised as a number. [561]
- Numeric conversions did not recognize changes in the string value properly in certain circumstances. [561]
- Attributes (such as `:shared`) didn't work with `our()`.
- `our()` variables will not cause bogus "Variable will not stay shared" warnings. [561]
- "our" variables of the same name declared in two sibling blocks resulted in bogus warnings about "redeclaration" of the variables. The problem has been corrected. [561]
- `pack "Z"` now correctly terminates the string with `"\0"`.
- Fix password routines which in some shadow password platforms (e.g. HP-UX) caused `getpwent()` to return every other entry.
- The `PERL5OPT` environment variable (for passing command line arguments to Perl) didn't work for more than a single group of options. [561]
- `PERL5OPT` with embedded spaces didn't work.

- `printf()` no longer resets the numeric locale to "C".
- `qw(a\\b)` now parses correctly as `'a\\b'`: that is, as three characters, not four. [561]
- `pos()` did not return the correct value within `s///ge` in earlier versions. This is now handled correctly. [561]
- Printing quads (64-bit integers) with `printf/sprintf` now works without the `q L ll` prefixes (assuming you are on a quad-capable platform).
- Regular expressions on references and overloaded scalars now work. [561+]
- Right-hand side magic (GMAGIC) could in many cases such as string concatenation be invoked too many times.
- `scalar()` now forces scalar context even when used in void context.
- SOCKS support is now much more robust.
- `sort()` arguments are now compiled in the right wantarray context (they were accidentally using the context of the `sort()` itself). The comparison block is now run in scalar context, and the arguments to be sorted are always provided list context. [561]
- Changed the POSIX character class `[[:space:]]` to include the (very rarely used) vertical tab character. Added a new POSIX-ish character class `[[:blank:]]` which stands for horizontal whitespace (currently, the space and the tab).
- The tainting behaviour of `sprintf()` has been rationalized. It does not taint the result of floating point formats anymore, making the behaviour consistent with that of string interpolation. [561]
- Some cases of inconsistent taint propagation (such as within hash values) have been fixed.
- The RE engine found in Perl 5.6.0 accidentally pessimised certain kinds of simple pattern matches. These are now handled better. [561]
- Regular expression debug output (whether through `use re 'debug'` or via `-Dr`) now looks better. [561]
- Multi-line matches like `"a\nxb\n" =~ /(?!\\A)x/m` were flawed. The bug has been fixed. [561]
- Use of `$&` could trigger a core dump under some situations. This is now avoided. [561]
- The regular expression captured submatches (`$1`, `$2`, ...) are now more consistently unset if the match fails, instead of leaving false data lying around in them. [561]
- `readline()` on files opened in "slurp" mode could return an extra "" (blank line) at the end in certain situations. This has been corrected. [561]
- Autovivification of symbolic references of special variables described in *perlvar* (as in `${ $num }`) was accidentally disabled. This works again now. [561]
- `Sys::Syslog` ignored the `LOG_AUTH` constant.
- `$AUTOLOAD`, `sort()`, `lock()`, and spawning subprocesses in multiple threads simultaneously are now thread-safe.
- `Tie::Array`'s `SPLICE` method was broken.
- Allow a read-only string on the left-hand side of a non-modifying `tr///`.
- If `STDERR` is tied, warnings caused by `warn` and `die` now correctly pass to it.
- Several Unicode fixes.

- BOMs (byte order marks) at the beginning of Perl files (scripts, modules) should now be transparently skipped. UTF-16 and UCS-2 encoded Perl files should now be read correctly.
- The character tables have been updated to Unicode 3.2.0.
- Comparing with utf8 data does not magically upgrade non-utf8 data into utf8. (This was a problem for example if you were mixing data from I/O and Unicode data: your output might have got magically encoded as UTF-8.)
- Generating illegal Unicode code points such as U+FFFE, or the UTF-16 surrogates, now also generates an optional warning.
- `IsAlnum`, `IsAlpha`, and `IsWord` now match titlecase.
- Concatenation with the `.` operator or via variable interpolation, eg, `substr`, `reverse`, `quotemeta`, the `x` operator, substitution with `s///`, single-quoted UTF8, should now work.
- The `tr///` operator now works. Note that the `tr///CU` functionality has been removed (but see `pack('U0', ...)`).
- `eval "v200"` now works.
- Perl 5.6.0 parsed `m/\x{ab}/` incorrectly, leading to spurious warnings. This has been corrected. [561]
- Zero entries were missing from the Unicode classes such as `IsDigit`.
- Large unsigned numbers (those above 2^{31}) could sometimes lose their unsignedness, causing bogus results in arithmetic operations. [561]
- The Perl parser has been stress tested using both random input and Markov chain input and the few found crashes and lockups have been fixed.

Platform Specific Changes and Fixes

- BSDI 4.*
Perl now works on post-4.0 BSD/OSes.
- All BSDs
Setting `$0` now works (as much as possible; see [perlvar](#) for details).
- Cygwin
Numerous updates; currently synchronised with Cygwin 1.3.10.
- Previously DYNIX/ptx had problems in its Configure probe for non-blocking I/O.
- EPOC
EPOC now better supported. See `README.epoc`. [561]
- FreeBSD 3.*
Perl now works on post-3.0 FreeBSDs.
- HP-UX
`README.hpux` updated; `Configure -Duse64bitall` now works; now uses HP-UX `malloc` instead of Perl `malloc`.
- IRIX
Numerous compilation flag and hint enhancements; accidental mixing of 32-bit and 64-bit libraries (a

doomed attempt) made much harder.

- **Linux**
 - Long doubles should now work (see INSTALL). [561]
 - Linux previously had problems related to `sockaddrlen` when using `accept()`, `recvfrom()` (in Perl: `recv()`), `getpeername()`, and `getsockname()`.
- **Mac OS Classic**

Compilation of the standard Perl distribution in Mac OS Classic should now work if you have the Metrowerks development environment and the missing Mac-specific toolkit bits. Contact the macperl mailing list for details.
- **MPE/iX**

MPE/iX update after Perl 5.6.0. See README.mpeix. [561]
- **NetBSD/threads**: try installing the GNU pth (should be in the packages collection, or <http://www.gnu.org/software/pth/>), and Configure with `-Duseithreads`.
- **NetBSD/sparc**

Perl now works on NetBSD/sparc.
- **OS/2**

Now works with `usethreads` (see INSTALL). [561]
- **Solaris**

64-bitness using the Sun Workshop compiler now works.
- **Stratus VOS**

The native build method requires at least VOS Release 14.5.0 and GNU C++/GNU Tools 2.0.1 or later. The Perl `pack` function now maps overflowed values to `+infinity` and underflowed values to `-infinity`.
- **Tru64 (aka Digital UNIX, aka DEC OSF/1)**

The operating system version letter now recorded in `$Config{osvers}`. Allow compiling with `gcc` (previously explicitly forbidden). Compiling with `gcc` still not recommended because buggy code results, even with `gcc 2.95.2`.
- **Unicos**

Fixed various alignment problems that lead into core dumps either during build or later; no longer dies on math errors at runtime; now using full quad integers (64 bits), previously was using only 46 bit integers for speed.
- **VMS**

See `and` for important changes not otherwise listed here.

`chdir()` now works better despite a CRT bug; now works with `MULTIPLICITY` (see INSTALL); now works with Perl's `malloc`.

The tainting of `%ENV` elements via `keys` or `values` was previously unimplemented. It now works as documented.

The `waitpid` emulation has been improved. The worst bug (now fixed) was that a `pid` of `-1` would cause a wildcard search of all processes on the system.

POSIX-style signals are now emulated much better on VMS versions prior to 7.0.

The `system` function and backticks operator have improved functionality and better error handling. [561]

File access tests now use current process privileges rather than the user's default privileges, which could sometimes result in a mismatch between reported access and actual access. This improvement is only available on VMS v6.0 and later.

There is a new `kill` implementation based on `sys$sigprc` that allows older VMS systems (pre-7.0) to use `kill` to send signals rather than simply force exit. This implementation also allows later systems to call `kill` from within a signal handler.

Iterative logical name translations are now limited to 10 iterations in imitation of `SHOW LOGICAL` and other OpenVMS facilities.

- Windows
 - Signal handling now works better than it used to. It is now implemented using a Windows message loop, and is therefore less prone to random crashes.
 - `fork()` emulation is now more robust, but still continues to have a few esoteric bugs and caveats. See [perlfork](#) for details. [561+]
 - A failed (pseudo)fork now returns `undef` and sets `errno` to `EAGAIN`. [561]
 - The following modules now work on Windows:


```
ExtUtils::Embed      [561]
IO::Pipe
IO::Poll
Net::Ping
```
 - `IO::File::new_tmpfile()` is no longer limited to 32767 invocations per-process.
 - Better `chdir()` return value for a non-existent directory.
 - Compiling perl using the 64-bit Platform SDK tools is now supported.
 - The `Win32::SetChildShowWindow()` builtin can be used to control the visibility of windows created by child processes. See [Win32](#) for details.
 - Non-blocking waits for child processes (or pseudo-processes) are supported via `waitpid($pid, &POSIX::WNOHANG)`.
 - The behavior of `system()` with multiple arguments has been rationalized. Each unquoted argument will be automatically quoted to protect whitespace, and any existing whitespace in the arguments will be preserved. This improves the portability of `system(@args)` by avoiding the need for Windows `cmd` shell specific quoting in perl programs.

Note that this means that some scripts that may have relied on earlier buggy behavior may no longer work correctly. For example, `system("nmake /nologo", @args)` will now attempt to run the file `nmake /nologo` and will fail when such a file isn't found. On the other hand, perl will now execute code such as `system("c:/Program Files/MyApp/foo.exe", @args)` correctly.
 - The perl header files no longer suppress common warnings from the Microsoft Visual C++ compiler. This means that additional warnings may now show up when compiling XS code.
 - Borland C++ v5.5 is now a supported compiler that can build Perl. However, the generated binaries continue to be incompatible with those generated by the other supported compilers (GCC and Visual C++). [561]

- Duping socket handles with `open(F, "&MYSOCK")` now works under Windows 9x. [561]
- Current directory entries in `%ENV` are now correctly propagated to child processes. [561]
- New `%ENV` entries now propagate to subprocesses. [561]
- `Win32::GetCwd()` correctly returns `C:\` instead of `C:` when at the drive root. Other bugs in `chdir()` and `Cwd::cwd()` have also been fixed. [561]
- The makefiles now default to the features enabled in ActiveState ActivePerl (a popular Win32 binary distribution). [561]
- HTML files will now be installed in `c:\perl\html` instead of `c:\perl\lib\pod\html`
- `REG_EXPAND_SZ` keys are now allowed in registry settings used by perl. [561]
- Can now `send()` from all threads, not just the first one. [561]
- `ExtUtils::MakeMaker` now uses `$ENV{LIB}` to search for libraries. [561]
- Less stack reserved per thread so that more threads can run concurrently. (Still 16M per thread.) [561]
- `< File::Spec::tmpdir()` now prefers `C:/temp` over `/tmp` (works better when perl is running as service).
- Better UNC path handling under `ithreads`. [561]
- `wait()`, `waitpid()`, and backticks now return the correct exit status under Windows 9x. [561]
- A socket handle leak in `accept()` has been fixed. [561]

New or Changed Diagnostics

Please see [perldiag](#) for more details.

- Ambiguous range in the transliteration operator (like `a-z-9`) now gives a warning.
- `chdir("")` and `chdir(undef)` now give a deprecation warning because they cause a possible unintentional `chdir` to the home directory. Say `chdir()` if you really mean that.
- Two new debugging options have been added: if you have compiled your Perl with debugging, you can use the `-DT` [561] and `-DR` options to trace tokenising and to add reference counts to displaying variables, respectively.
- The lexical warnings category "deprecated" is no longer a sub-category of the "syntax" category. It is now a top-level category in its own right.
- Unadorned `dump()` will now give a warning suggesting to use explicit `CORE::dump()` if that's what really is meant.
- The "Unrecognized escape" warning has been extended to include `\8`, `\9`, and `_`. There is no need to escape any of the `\w` characters.
- All regular expression compilation error messages are now hopefully easier to understand both because the error message now comes before the failed regex and because the point of failure is now clearly marked by a `<- HERE` marker.
- Various I/O (and socket) functions like `binmode()`, `close()`, and so forth now more consistently warn if they are used illogically either on a yet unopened or on an already closed filehandle (or socket).
- Using `lstat()` on a filehandle now gives a warning. (It's a non-sensical thing to do.)

- The `-M` and `-m` options now warn if you didn't supply the module name.
- If you in `use` specify a required minimum version, modules matching the name and but not defining a `$VERSION` will cause a fatal failure.
- Using negative offset for `vec()` in lvalue context is now a warnable offense.
- Odd number of arguments to `overload::constant` now elicits a warning.
- Odd number of elements to in anonymous hash now elicits a warning.
- The various "opened only for", "on closed", "never opened" warnings drop the `main::` prefix for filehandles in the `main` package, for example `STDIN` instead of `main::STDIN`.
- Subroutine prototypes are now checked more carefully, you may get warnings for example if you have used non-prototype characters.
- If an attempt to use a (non-blessed) reference as an array index is made, a warning is given.
- `push @a;` and `unshift @a;` (with no values to push or unshift) now give a warning. This may be a problem for generated and eval'd code.
- If you try to *pack* a number less than 0 or larger than 255 using the "C" format you will get an optional warning. Similarly for the "c" format and a number less than -128 or more than 127.
- `pack P` format now demands an explicit size.
- `unpack w` now warns of unterminated compressed integers.
- Warnings relating to the use of `PerlIO` have been added.
- Certain regex modifiers such as `(?o)` make sense only if applied to the entire regex. You will get an optional warning if you try to do otherwise.
- Variable length lookbehind has not yet been implemented, trying to use it will tell that.
- Using arrays or hashes as references (e.g. `< %foo->{bar}`) has been deprecated for a while. Now you will get an optional warning.
- Warnings relating to the use of the new restricted hashes feature have been added.
- Self-ties of arrays and hashes are not supported and fatal errors will happen even at an attempt to do so.
- Using `sort` in scalar context now issues an optional warning. This didn't do anything useful, as the sort was not performed.
- Using the `/g` modifier in `split()` is meaningless and will cause a warning.
- Using `splice()` past the end of an array now causes a warning.
- Malformed Unicode encodings (UTF-8 and UTF-16) cause a lot of warnings, and doestrying to use UTF-16 surrogates (which are unimplemented).
- Trying to use Unicode characters on an I/O stream without marking the stream's encoding (using `open()` or `binmode()`) will cause "Wide character" warnings.
- Use of v-strings in `use/require` causes a (backward) portability warning.
- Warnings relating to the use interpreter threads and their shared data have been added.

Changed Internals

- `PerlIO` is now the default.

- `perlapi.pod` (a companion to `perlguts`) now attempts to document the internal API.
- You can now build a really minimal perl called `microperl`. Building `microperl` does not require even running `Configure`; `make -f Makefile.micro` should be enough. Beware: `microperl` makes many assumptions, some of which may be too bold; the resulting executable may crash or otherwise misbehave in wondrous ways. For careful hackers only.
- Added `rsignal()`, `whichsig()`, `do_join()`, `op_clear`, `op_null`, `ptr_table_clear()`, `ptr_table_free()`, `sv_setref_uv()`, and several UTF-8 interfaces to the publicised API. For the full list of the available APIs see [perlapi](#).
- Made possible to propagate customised exceptions via `croak()`ing.
- Now `xsubs` can have attributes just like `subs`. (Well, at least the built-in attributes.)
- `dTHR` and `djSP` have been obsoleted; the former removed (because it's a no-op) and the latter replaced with `dSP`.
- `PERL_OBJECT` has been completely removed.
- The `MAGIC` constants (e.g. `'P'`) have been macrofied (e.g. `PERL_MAGIC_TIED`) for better source code readability and maintainability.
- The regex compiler now maintains a structure that identifies nodes in the compiled bytecode with the corresponding syntactic features of the original regex expression. The information is attached to the new `offsets` member of the `struct regexp`. See [perldebugs](#) for more complete information.
- The C code has been made much more `gcc -Wall` clean. Some warning messages still remain in some platforms, so if you are compiling with `gcc` you may see some warnings about dubious practices. The warnings are being worked on.
- *`perly.c`*, *`sv.c`*, and *`sv.h`* have now been extensively commented.
- Documentation on how to use the Perl source repository has been added to *`Porting/repository.pod`*.
- There are now several profiling make targets.

Security Vulnerability Closed [561]

(This change was already made in 5.7.0 but bears repeating here.) (5.7.0 came out before 5.6.1: the development branch 5.7 released earlier than the maintenance branch 5.6)

A potential security vulnerability in the optional `suidperl` component of Perl was identified in August 2000. `suidperl` is neither built nor installed by default. As of November 2001 the only known vulnerable platform is Linux, most likely all Linux distributions. CERT and various vendors and distributors have been alerted about the vulnerability. See <http://www.cpan.org/src/5.0/sperl-2000-08-05/sperl-2000-08-05.txt> for more information.

The problem was caused by Perl trying to report a suspected security exploit attempt using an external program, `/bin/mail`. On Linux platforms the `/bin/mail` program had an undocumented feature which when combined with `suidperl` gave access to a root shell, resulting in a serious compromise instead of reporting the exploit attempt. If you don't have `/bin/mail`, or if you have 'safe setuid scripts', or if `suidperl` is not installed, you are safe.

The exploit attempt reporting feature has been completely removed from Perl 5.8.0 (and the maintenance release 5.6.1, and it was removed also from all the Perl 5.7 releases), so that particular vulnerability isn't there anymore. However, further security vulnerabilities are, unfortunately, always possible. The `suidperl` functionality is most probably going to be removed in Perl 5.10. In any case, `suidperl` should only be used by security experts who know exactly what they are doing and why they are using `suidperl` instead of some other solution such as `sudo` (see <http://www.courtesan.com/sudo/>).

New Tests

Several new tests have been added, especially for the *lib* and *ext* subsections. There are now about 69 000 individual tests (spread over about 700 test scripts), in the regression suite (5.6.1 has about 11 700 tests, in 258 test scripts). The exact numbers depend on the platform and Perl configuration used. Many of the new tests are of course introduced by the new modules, but still in general Perl is now more thoroughly tested.

Because of the large number of tests, running the regression suite will take considerably longer time than it used to: expect the suite to take up to 4–5 times longer to run than in perl 5.6. On a really fast machine you can hope to finish the suite in about 6–8 minutes (wallclock time).

The tests are now reported in a different order than in earlier Perls. (This happens because the test scripts from under *t/lib* have been moved to be closer to the library/extension they are testing.)

Known Problems

The Compiler Suite Is Still Very Experimental

The compiler suite is slowly getting better but it continues to be highly experimental. Use in production environments is discouraged.

Localising Tied Arrays and Hashes Is Broken

```
local %tied_array;
```

doesn't work as one would expect: the old value is restored incorrectly. This will be changed in a future release, but we don't know yet what the new semantics will exactly be. In any case, the change will break existing code that relies on the current (ill-defined) semantics, so just avoid doing this in general.

Building Extensions Can Fail Because Of Largefiles

Some extensions like *mod_perl* are known to have issues with 'largefiles', a change brought by Perl 5.6.0 in which file offsets default to 64 bits wide, where supported. Modules may fail to compile at all, or they may compile and work incorrectly. Currently, there is no good solution for the problem, but *Configure* now provides appropriate non-largefile *ccflags*, *ldflags*, *libswanted*, and *libs* in the *%Config* hash (e.g., *\$Config{ccflags_nolargefiles}*) so the extensions that are having problems can try configuring themselves without the largefile-ness. This is admittedly not a clean solution, and the solution may not even work at all. One potential failure is whether one can (or, if one can, whether it's a good idea to) link together at all binaries with different ideas about file offsets; all this is platform-dependent.

Modifying \$_ Inside for(..)

```
for (1..5) { $_++ }
```

works without complaint. It shouldn't. (You should be able to modify only *lvalue* elements inside the loops.) You can see the correct behaviour by replacing the *1..5* with *1, 2, 3, 4, 5*.

mod_perl 1.26 Doesn't Build With Threaded Perl

Use *mod_perl* 1.27 or higher.

lib/ftmp-security tests warn 'system possibly insecure'

Don't panic. Read the 'make test' section of *INSTALL* instead.

libwww-perl (LWP) fails base/date #51

Use *libwww-perl* 5.65 or later.

PDL failing some tests

Use *PDL* 2.3.4 or later.

Perl_get_sv

You may get errors like 'Undefined symbol "Perl_get_sv"' or "can't resolve symbol 'Perl_get_sv'", or the symbol may be "Perl_sv_2pv". This probably means that you are trying to use an older shared Perl library (or extensions linked with such) with Perl 5.8.0 executable. Perl used to have such a subroutine, but that is no more the case. Check your shared library path, and any shared Perl libraries in those directories.

Sometimes this problem may also indicate a partial Perl 5.8.0 installation, see [for an example](#) and how to deal with it.

Self-tying Problems

Self-tying of arrays and hashes is broken in rather deep and hard-to-fix ways. As a stop-gap measure to avoid people from getting frustrated at the mysterious results (core dumps, most often), it is forbidden for now (you will get a fatal error even from an attempt).

A change to self-tying of globs has caused them to be recursively referenced (see:

[Two-Phased Garbage Collection in perlobj](#)). You will now need an explicit untie to destroy a self-tied glob. This behaviour may be fixed at a later date.

Self-tying of scalars and IO thingies works.

ext/threads/t/libc

If this test fails, it indicates that your libc (C library) is not threadsafe. This particular test stress tests the `localtime()` call to find out whether it is threadsafe. See [perlthrtut](#) for more information.

Failure of Thread (5.005-style) tests

Note that support for 5.005-style threading is deprecated, experimental and practically unsupported. In 5.10, it is expected to be removed. You should migrate your code to ithreads.

The following tests are known to fail due to fundamental problems in the 5.005 threading implementation. These are not new failures—Perl 5.005_0x has the same bugs, but didn't have these tests.

../ext/B/t/xref.t	255 65280	14	12	85.71%	3-14
../ext/List/Util/t/first.t	255 65280	7	4	57.14%	2 5-7
../lib/English.t	2 512	54	2	3.70%	2-3
../lib/FileCache.t		5	1	20.00%	5
../lib/Filter/Simple/t/data.t		6	3	50.00%	1-3
../lib/Filter/Simple/t/filter_only.		9	3	33.33%	1-2 5
../lib/Math/BigInt/t/bare_mbf.t		1627	4	0.25%	8 11 1626-1627
../lib/Math/BigInt/t/bigfltpm.t		1629	4	0.25%	10 13 1628-1629
../lib/Math/BigInt/t/sub_mbf.t		1633	4	0.24%	8 11 1632-1633
../lib/Math/BigInt/t/with_sub.t		1628	4	0.25%	9 12 1627-1628
../lib/Tie/File/t/31_autodefer.t	255 65280	65	32	49.23%	34-65
../lib/autouse.t		10	1	10.00%	4
op/flip.t		15	1	6.67%	15

These failures are unlikely to get fixed as 5.005-style threads are considered fundamentally broken. (Basically what happens is that competing threads can corrupt shared global state, one good example being regular expression engine's state.)

Timing problems

The following tests may fail intermittently because of timing problems, for example if the system is heavily loaded.

```
t/op/alarm.t
ext/Time/HiRes/HiRes.t
lib/Benchmark.t
lib/Memoize/t/expmod_t.t
lib/Memoize/t/speed.t
```

In case of failure please try running them manually, for example

```
./perl -Ilib ext/Time/HiRes/HiRes.t
```

Tied/Magical Array/Hash Elements Do Not Autovivify

For normal arrays `$foo = \ $bar[1]` will assign `undef` to `$bar[1]` (assuming that it didn't exist before), but for tied/magical arrays and hashes such autovivification does not happen because there is currently no way to catch the reference creation. The same problem affects slicing over non-existent indices/keys of a tied/magical array/hash.

Unicode in package/class and subroutine names does not work

One can have Unicode in identifier names, but not in package/class or subroutine names. While some limited functionality towards this does exist as of Perl 5.8.0, that is more accidental than designed; use of Unicode for the said purposes is unsupported.

One reason of this unfinishedness is its (currently) inherent unportability: since both package names and subroutine names may need to be mapped to file and directory names, the Unicode capability of the filesystem becomes important—and there unfortunately aren't portable answers.

Platform Specific Problems

AIX

- If using the AIX native make command, instead of just "make" issue "make all". In some setups the former has been known to spuriously also try to run "make install". Alternatively, you may want to use GNU make.
- In AIX 4.2, Perl extensions that use C++ functions that use statics may have problems in that the statics are not getting initialized. In newer AIX releases, this has been solved by linking Perl with the `libC_r` library, but unfortunately in AIX 4.2 the said library has an obscure bug where the various functions related to time (such as `time()` and `gettimeofday()`) return broken values, and therefore in AIX 4.2 Perl is not linked against `libC_r`.

- vac 5.0.0.0 May Produce Buggy Code For Perl

The AIX C compiler vac version 5.0.0.0 may produce buggy code, resulting in a few random tests failing when run as part of "make test", but when the failing tests are run by hand, they succeed. We suggest upgrading to at least vac version 5.0.1.0, that has been known to compile Perl correctly. "`lspp -L|grep vac.C`" will tell you the vac version. See `README.aix`.

- If building threaded Perl, you may get compilation warning from `pp_sys.c`:

```
"pp_sys.c", line 4651.39: 1506-280 (W) Function argument assignment between typ
```

This is harmless; it is caused by the `getnetbyaddr()` and `getnetbyaddr_r()` having slightly different types for their first argument.

Alpha systems with old gccs fail several tests

If you see `op/pack`, `op/pat`, `op/regexp`, or `ext/Storable` tests failing in a Linux/alpha or *BSD/Alpha, it's probably time to upgrade your gcc. gccs prior to 2.95.3 are definitely not good enough, and gcc 3.1 may be even better. (RedHat Linux/alpha with gcc 3.1 reported no problems, as did Linux 2.4.18 with gcc 2.95.4.) (In Tru64, it is preferable to use the bundled C compiler.)

AmigaOS

Perl 5.8.0 doesn't build in AmigaOS. It broke at some point during the `ithreads` work and we could not find Amiga experts to unbreak the problems. Perl 5.6.1 still works for AmigaOS (as does the the 5.7.2 development release).

BeOS

The following tests fail on 5.8.0 Perl in BeOS Personal 5.03:

```
t/op/lfs.....FAILED at test 17
t/op/magic.....FAILED at test 24
ext/Fcntl/t/syslfs.....FAILED at test 17
ext/File/Glob/t/basic.....FAILED at test 3
```

```
ext/POSIX/t/sigaction.....FAILED at test 13
ext/POSIX/t/waitpid.....FAILED at test 1
```

See [perlbeos](#) (README.beos) for more details.

Cygwin "unable to remap"

For example when building the Tk extension for Cygwin, you may get an error message saying "unable to remap". This is known problem with Cygwin, and a workaround is detailed in here:

<http://sources.redhat.com/ml/cygwin/2001-12/msg00894.html>

Cygwin ndbm tests fail on FAT

One can build but not install (or test the build of) the NDBM_File on FAT filesystems. Installation (or build) on NTFS works fine. If one attempts the test on a FAT install (or build) the following failures are expected:

```
../ext/NDBM_File/ndbm.t      13  3328    71   59  83.10%  1-2  4  16-71
../ext/ODBM_File/odbm.t     255 65280   ??   ??      %   ??
../lib/AnyDBM_File.t        2   512    12    2  16.67%  1  4
../lib/Memoize/t/errors.t    0   139    11    5  45.45%  7-11
../lib/Memoize/t/tie_ndbm.t  13  3328     4    4 100.00%  1-4
run/fresh_perl.t            97    1    1.03%  91
```

NDBM_File fails and ODBM_File just coredumps.

DJGPP Failures

```
t/op/stat.....FAILED at test 29
lib/File/Find/t/find.....FAILED at test 1
lib/File/Find/t/taint.....FAILED at test 1
lib/h2xs.....FAILED at test 15
lib/Pod/t/eol.....FAILED at test 1
lib/Test/Harness/t/strap-analyze.....FAILED at test 8
lib/Test/Harness/t/test-harness.....FAILED at test 23
lib/Test/Simple/t/exit.....FAILED at test 1
```

The above failures are known as of 5.8.0 with native builds with long filenames, but there are a few more if running under dosemu because of limitations (and maybe bugs) of dosemu:

```
t/comp/cpp.....FAILED at test 3
t/op/inccode.....(crash)
```

and a few lib/ExtUtils tests, and several hundred Encode/t/Aliases.t failures that work fine with long filenames. So you really might prefer native builds and long filenames.

FreeBSD built with ithreads coredumps reading large directories

This is a known bug in FreeBSD 4.5's `readdir_r()`, it has been fixed in FreeBSD 4.6 (see [perlfreebsd](#) (README.freebsd)).

FreeBSD Failing locale Test 117 For ISO 8859-15 Locales

The ISO 8859-15 locales may fail the locale test 117 in FreeBSD. This is caused by the characters `\xFF` (y with diaeresis) and `\xBE` (Y with diaeresis) not behaving correctly when being matched case-insensitively. Apparently this problem has been fixed in the latest FreeBSD releases. (

<http://www.freebsd.org/cgi/query-pr.cgi?pr=34308>)

IRIX fails ext/List/Util/t/shuffle.t or Digest::MD5

IRIX with MIPSpro 7.3.1.2m or 7.3.1.3m compiler may fail the List::Util test `ext/List/Util/t/shuffle.t` by dumping core. This seems to be a compiler error since if compiled with gcc no core dump ensues, and no failures have been seen on the said test on any other platform.

Similarly, building the Digest::MD5 extension has been known to fail with "**** Termination code 139 (bu21)".

The cure is to drop optimization level (Configure `--Doptimize=-O2`).

HP-UX lib/posix Subtest 9 Fails When LP64-Configured

If perl is configured with `--Duse64bitall`, the successful result of the subtest 10 of lib/posix may arrive before the successful result of the subtest 9, which confuses the test harness so much that it thinks the subtest 9 failed.

Linux with glibc 2.2.5 fails t/op/int subtest #6 with `--Duse64bitint`

This is a known bug in the glibc 2.2.5 with long long integers. (
http://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=65612)

Linux With Sfifo Fails op/misc Test 48

No known fix.

Mac OS X

Please remember to set your environment variable `LC_ALL` to "C" (`setenv LC_ALL C`) before running "make test" to avoid a lot of warnings about the broken locales of Mac OS X.

The following tests are known to fail in Mac OS X 10.1.5 because of buggy (old) implementations of Berkeley DB included in Mac OS X:

Failed Test	Stat	Wstat	Total	Fail	Failed	List of Failed
../ext/DB_File/t/db-btree.t	0	11	??	??	%	??
../ext/DB_File/t/db-recno.t			149	3	2.01%	61 63 65

If you are building on a UFS partition, you will also probably see `t/op/stat.t` subtest #9 fail. This is caused by Darwin's UFS not supporting inode change time.

Also the `ext/POSIX/t/posix.t` subtest #10 fails but it is skipped for now because the failure is Apple's fault, not Perl's (blocked signals are lost).

If you Configure with `ithreads`, `ext/threads/t/libc.t` will fail. Again, this is not Perl's fault— the `libc` of Mac OS X is not threadsafe (in this particular test, the `localtime()` call is found to be threadunsafe.)

Mac OS X dyld undefined symbols

If after installing Perl 5.8.0 you are getting warnings about missing symbols, for example

```
dyld: perl Undefined symbols
  _perl_sv_2pv
  _perl_get_sv
```

you probably have an old pre-Perl-5.8.0 installation (or parts of one) in `/Library/Perl` (the undefined symbols used to exist in pre-5.8.0 Perls). It seems that for some reason "make install" doesn't always completely overwrite the files in `/Library/Perl`. You can move the old Perl shared library out of the way like this:

```
cd /Library/Perl/darwin/CORE
mv libperl.dylib libperlold.dylib
```

and then reissue "make install". Note that the above of course is extremely disruptive for anything using the `/usr/local/bin/perl`. If that doesn't help, you may have to try removing all the `.bundle` files from beneath `/Library/Perl`, and again "make install"-ing.

OS/2 Test Failures

The following tests are known to fail on OS/2 (for clarity only the failures are shown, not the full error messages):

../lib/ExtUtils/t/Mkbootstrap.t	1	256	18	1	5.56%	8
../lib/ExtUtils/t/Packlist.t	1	256	34	1	2.94%	17
../lib/ExtUtils/t/basic.t	1	256	17	1	5.88%	14

lib/os2_process.t	2	512	227	2	0.88%	174	209
lib/os2_process_kid.t			227	2	0.88%	174	209
lib/rx_cmpprt.t	255	65280	18	3	16.67%	16	18

op/sprintf tests 91, 129, and 130

The op/sprintf tests 91, 129, and 130 are known to fail on some platforms. Examples include any platform using sfio, and Compaq/Tandem's NonStop-UX.

Test 91 is known to fail on QNX6 (nto), because `sprintf '%e', 0` incorrectly produces `0.000000e+0` instead of `0.000000e+00`.

For tests 129 and 130, the failing platforms do not comply with the ANSI C Standard: lines 19ff on page 134 of ANSI X3.159 1989, to be exact. (They produce something other than "1" and "-1" when formatting 0.6 and -0.6 using the printf format "%.0f"; most often, they produce "0" and "-0".)

Solaris 2.5

In case you are still using Solaris 2.5 (aka SunOS 5.5), you may experience failures (the test core dumping) in lib/locale.t. The suggested cure is to upgrade your Solaris.

Solaris x86 Fails Tests With -Duse64bitint

The following tests are known to fail in Solaris x86 with Perl configured to use 64 bit integers:

```
ext/Data/Dumper/t/dumper.....FAILED at test 268
ext/Devel/Peek/Peek.....FAILED at test 7
```

SUPER-UX (NEC SX)

The following tests are known to fail on SUPER-UX:

```
op/64bitint.....FAILED tests 29-30, 32-33, 35-36
op/arith.....FAILED tests 128-130
op/pack.....FAILED tests 25-5625
op/pow.....
op/taint.....# msgsnd failed
../ext/IO/lib/IO/t/io_poll.....FAILED tests 3-4
../ext/IPC/SysV/ipcsysv.....FAILED tests 2, 5-6
../ext/IPC/SysV/t/msg.....FAILED tests 2, 4-6
../ext/Socket/socketpair.....FAILED tests 12
../lib/IPC/SysV.....FAILED tests 2, 5-6
../lib/warnings.....FAILED tests 115-116, 118-119
```

The op/pack failure ("Cannot compress negative numbers at op/pack.t line 126") is serious but as of yet unsolved. It points at some problems with the signedness handling of the C compiler, as do the 64bitint, arith, and pow failures. Most of the rest point at problems with SysV IPC.

Term::ReadKey not working on Win32

Use Term::ReadKey 2.20 or later.

UNICOS/mk

- During Configure, the test

Guessing which symbols your C compiler and preprocessor define...

will probably fail with error messages like

```
CC-20 cc: ERROR File = try.c, Line = 3
The identifier "bad" is undefined.
```

```
bad switch yylook 79bad switch yylook 79bad switch yylook 79bad switch yylo
^
```

```
CC-65 cc: ERROR File = try.c, Line = 3
```

A semicolon is expected at this point.

This is caused by a bug in the `awk` utility of UNICOS/mk. You can ignore the error, but it does cause a slight problem: you cannot fully benefit from the `h2ph` utility (see [h2ph](#)) that can be used to convert C headers to Perl libraries, mainly used to be able to access from Perl the constants defined using C preprocessor, `cpp`. Because of the above error, parts of the converted headers will be invisible. Luckily, these days the need for `h2ph` is rare.

- If building Perl with interpreter threads (`ithreads`), the `getgrent()`, `getgrnam()`, and `getgrgid()` functions cannot return the list of the group members due to a bug in the multithreaded support of UNICOS/mk. What this means is that in list context the functions will return only three values, not four.

UTS

There are a few known test failures, see [perluts](#) (README.uts).

VOS (Stratus)

When Perl is built using the native build process on VOS Release 14.5.0 and GNU C++/GNU Tools 2.0.1, all attempted tests either pass or result in TODO (ignored) failures.

VMS

There should be no reported test failures with a default configuration, though there are a number of tests marked TODO that point to areas needing further debugging and/or porting work.

Win32

In multi-CPU boxes, there are some problems with the I/O buffering: some output may appear twice.

XML::Parser not working

Use XML::Parser 2.31 or later.

z/OS (OS/390)

z/OS has rather many test failures but the situation is actually much better than it was in 5.6.0; it's just that so many new modules and tests have been added.

Failed Test	Stat	Wstat	Total	Fail	Failed	List of Failed
../ext/Data/Dumper/t/dumper.t			357	8	2.24%	311 314 325 327 331 333 337 339
../ext/IO/lib/IO/t/io_unix.t			5	4	80.00%	2-5
../ext/Storable/t/downgrade.t	12	3072	169	12	7.10%	14-15 46-47 78-79 110-111 150 161
../lib/ExtUtils/t/Constant.t	121	30976	48	48	100.00%	1-48
../lib/ExtUtils/t/Embed.t			9	9	100.00%	1-9
op/pat.t			922	7	0.76%	665 776 785 832- 834 845
op/sprintf.t			224	3	1.34%	98 100 136
op/tr.t			97	5	5.15%	63 71-74
uni/fold.t			780	6	0.77%	61 169 196 661 710-711

The failures in `dumper.t` and `downgrade.t` are problems in the tests, those in `io_unix` and `sprintf` are problems in the USS (UDP sockets and `printf` formats). The `pat`, `tr`, and `fold` failures are genuine Perl problems caused by EBCDIC (and in the `pat` and `fold` cases, combining that with Unicode). The `Constant` and `Embed` are probably problems in the tests (since they test Perl's ability to build extensions, and that seems to be working reasonably well.)

Unicode Support on EBCDIC Still Spotty

Though mostly working, Unicode support still has problem spots on EBCDIC platforms. One such known spot are the `\p{}` and `\P{}` regular expression constructs for code points less than 256: the `pP` are testing for Unicode code points, not knowing about EBCDIC.

Seen In Perl 5.7 But Gone Now

`Time::Piece` (previously known as `Time::Object`) was removed because it was felt that it didn't have enough value in it to be a core module. It is still a useful module, though, and is available from the CPAN.

Perl 5.8 unfortunately does not build anymore on AmigaOS; this broke accidentally at some point. Since there are not that many Amiga developers available, we could not get this fixed and tested in time for 5.8.0. Perl 5.6.1 still works for AmigaOS (as does the the 5.7.2 development release).

The `PerlIO::Scalar` and `PerlIO::Via` (capitalised) were renamed as `PerlIO::scalar` and `PerlIO::via` (all lowercase) just before 5.8.0. The main rationale was to have all core PerlIO layers to have all lowercase names. The "plugins" are named as usual, for example `PerlIO::via::QuotedPrint`.

The `threads::shared::queue` and `threads::shared::semaphore` were renamed as `Thread::Queue` and `Thread::Semaphore` just before 5.8.0. The main rationale was to have thread modules to obey normal naming, `Thread::` (the `threads` and `threads::shared` themselves are more pragma-like, they affect compile-time, so they stay lowercase).

Reporting Bugs

If you find what you think is a bug, you might check the articles recently posted to the `comp.lang.perl.misc` newsgroup and the perl bug database at <http://bugs.perl.org/> . There may also be information at <http://www.perl.com/> , the Perl Home Page.

If you believe you have an unreported bug, please run the **perlbug** program included with your release. Be sure to trim your bug down to a tiny but sufficient test case. Your bug report, along with the output of `perl -V`, will be sent off to `perlbug@perl.org` to be analysed by the Perl porting team.

SEE ALSO

The *Changes* file for exhaustive details on what changed.

The *INSTALL* file for how to build Perl.

The *README* file for general stuff.

The *Artistic* and *Copying* files for copyright information.

HISTORY

Written by Jarkko Hietaniemi <jhi@iki.fi>.

NAME

perldiag – various Perl diagnostics

DESCRIPTION

These messages are classified as follows (listed in increasing order of desperation):

- (W) A warning (optional).
- (D) A deprecation (optional).
- (S) A severe warning (default).
- (F) A fatal error (trappable).
- (P) An internal error you should never see (trappable).
- (X) A very fatal error (nontrappable).
- (A) An alien error message (not generated by Perl).

The majority of messages from the first three classifications above (W, D & S) can be controlled using the `warnings` pragma.

If a message can be controlled by the `warnings` pragma, its warning category is included with the classification letter in the description below.

Optional warnings are enabled by using the `warnings` pragma or the `-w` and `-W` switches. Warnings may be captured by setting `$SIG{__WARN__}` to a reference to a routine that will be called on each warning instead of printing it. See [perlvar](#).

Default warnings are always enabled unless they are explicitly disabled with the `warnings` pragma or the `-X` switch.

Trappable errors may be trapped using the `eval` operator. See [eval](#). In almost all cases, warnings may be selectively disabled or promoted to fatal errors using the `warnings` pragma. See [warnings](#).

The messages are in alphabetical order, without regard to upper or lower-case. Some of these messages are generic. Spots that vary are denoted with a `%s` or other `printf`-style escape. These escapes are ignored by the alphabetical order, as are all characters other than letters. To look up your message, just ignore anything that is not a letter.

A thread exited while %d other threads were still running

- (W) When using threaded Perl, a thread (not necessarily the main thread) exited while there were still other threads running. Usually it's a good idea to first collect the return values of the created threads by joining them, and only then exit from the main thread. See [threads](#).

accept () on closed socket %s

- (W closed) You tried to do an `accept` on a closed socket. Did you forget to check the return value of your `socket ()` call? See [accept](#).

Allocation too large: %lx

- (X) You can't allocate more than 64K on an MS-DOS machine.

'!' allowed only after types %s

- (F) The `'!'` is allowed in `pack ()` and `unpack ()` only after certain types. See [pack](#).

Ambiguous call resolved as CORE::: %s (), qualify as such or use &

- (W ambiguous) A subroutine you have declared has the same name as a Perl keyword, and you have used the name without qualification for calling one or the other. Perl decided to call the builtin because the subroutine is not imported.

To force interpretation as a subroutine call, either put an ampersand before the subroutine name, or qualify the name with its package. Alternatively, you can import the subroutine (or pretend that it's imported with the `use subs` pragma).

To silently interpret it as the Perl operator, use the `CORE:::` prefix on the operator (e.g.

`CORE::log($x)` or declare the subroutine to be an object method (see [Subroutine Attributes in perlsyn](#) or [attributes](#)).

Ambiguous range in transliteration operator

(F) You wrote something like `tr/a-z-0//` which doesn't mean anything at all. To include a `-` character in a transliteration, put it either first or last. (In the past, `tr/a-z-0//` was synonymous with `tr/a-y//`, which was probably not what you would have expected.)

Ambiguous use of `%s` resolved as `%s`

(W ambiguous)(S) You said something that may not be interpreted the way you thought. Normally it's pretty easy to disambiguate it by supplying a missing quote, operator, parenthesis pair or declaration.

`'` and `<` may not both be specified on command line

(F) An error peculiar to VMS. Perl does its own command line redirection, and found that STDIN was a pipe, and that you also tried to redirect STDIN using `<`. Only one STDIN stream to a customer, please.

`'` and `"` may not both be specified on command line

(F) An error peculiar to VMS. Perl does its own command line redirection, and thinks you tried to redirect stdout both to a file and into a pipe to another command. You need to choose one or the other, though nothing's stopping you from piping into a program or Perl script which 'splits' output into two streams, such as

```
open(OUT, ">$ARGV[0]") or die "Can't write to $ARGV[0]: $!";
while (<STDIN>) {
    print;
    print OUT;
}
close OUT;
```

Applying `%s` to `%s` will act on scalar(`%s`)

(W misc) The pattern match (`//`), substitution (`s///`), and transliteration (`tr///`) operators work on scalar values. If you apply one of them to an array or a hash, it will convert the array or hash to a scalar value — the length of an array, or the population info of a hash — and then work on that scalar value. This is probably not what you meant to do. See [grep](#) and [map](#) for alternatives.

Args must match `#!` line

(F) The setuid emulator requires that the arguments Perl was invoked with match the arguments specified on the `#!` line. Since some systems impose a one-argument limit on the `#!` line, try combining switches; for example, turn `-w -U` into `-wU`.

Arg too short for `msgsnd`

(F) `msgsnd()` requires a string at least as long as `sizeof(long)`.

`%s` argument is not a HASH or ARRAY element

(F) The argument to `exists()` must be a hash or array element, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
```

`%s` argument is not a HASH or ARRAY element or slice

(F) The argument to `delete()` must be either a hash or array element, such as:

```
$foo{$bar}
$ref->{"susie"}[12]
```

or a hash or array slice, such as:

```
@foo[$bar, $baz, $xyzyz]
@{$ref->[12]}{"susie", "queue"}
```

%s argument is not a subroutine name

(F) The argument to `exists()` for `exists &sub` must be a subroutine name, and not a subroutine call. `exists &sub()` will generate this error.

Argument "%s" isn't numeric%s

(W numeric) The indicated string was fed as an argument to an operator that expected a numeric value instead. If you're fortunate the message will identify which operator was so unfortunate.

Array @%s missing the @ in argument %d of %s()

(D deprecated) Really old Perl let you omit the @ on array names in some spots. This is now heavily deprecated.

assertion botched: %s

(P) The malloc package that comes with Perl had an internal failure.

Assertion failed: file "%s"

(P) A general assertion failed. The file in question must be examined.

Assignment to both a list and a scalar

(F) If you assign to a conditional operator, the 2nd and 3rd arguments must either both be scalars or both be lists. Otherwise Perl won't know which context to supply to the right side.

Attempt to access disallowed key '%s' in a restricted hash

(F) The failing code has attempted to get or set a key which is not in the current set of allowed keys of a restricted hash.

Attempt to clear a restricted hash

(F) It is currently not allowed to clear a restricted hash, even if the new hash would contain the same keys as before. This may change in the future.

Attempt to delete readonly key '%s' from a restricted hash

(F) The failing code attempted to delete a key whose value has been declared readonly from a restricted hash.

Attempt to delete disallowed key '%s' from a restricted hash

(F) The failing code attempted to delete from a restricted hash a key which is not in its key set.

Attempt to bless into a reference

(F) The `CLASSNAME` argument to the `bless()` operator is expected to be the name of the package to bless the resulting object into. You've supplied instead a reference to something: perhaps you wrote

```
bless $self, $proto;
```

when you intended

```
bless $self, ref($proto) || $proto;
```

If you actually want to bless into the stringified version of the reference supplied, you need to stringify it yourself, for example by:

```
bless $self, "$proto";
```

Attempt to free non-arena SV: 0x%lx

(P internal) All SV objects are supposed to be allocated from arenas that will be garbage collected on exit. An SV was discovered to be outside any of those arenas.

Attempt to free nonexistent shared string

(P internal) Perl maintains a reference counted internal table of strings to optimize the storage and access of hash keys and other strings. This indicates someone tried to decrement the reference count of a string that can no longer be found in the table.

Attempt to free temp prematurely

(W debugging) Mortalized values are supposed to be freed by the `free_tmps()` routine. This indicates that something else is freeing the SV before the `free_tmps()` routine gets a chance, which means that the `free_tmps()` routine will be freeing an unreferenced scalar when it does try to free it.

Attempt to free unreferenced glob pointers

(P internal) The reference counts got screwed up on symbol aliases.

Attempt to free unreferenced scalar

(W internal) Perl went to decrement the reference count of a scalar to see if it would go to 0, and discovered that it had already gone to 0 earlier, and should have been freed, and in fact, probably was freed. This could indicate that `SvREFCNT_dec()` was called too many times, or that `SvREFCNT_inc()` was called too few times, or that the SV was mortalized when it shouldn't have been, or that memory has been corrupted.

Attempt to join self

(F) You tried to join a thread from within itself, which is an impossible task. You may be joining the wrong thread, or you may need to move the `join()` to some other thread.

Attempt to pack pointer to temporary value

(W pack) You tried to pass a temporary value (like the result of a function, or a computed expression) to the "p" `pack()` template. This means the result contains a pointer to a location that could become invalid anytime, even before the end of the current statement. Use literals or global values as arguments to the "p" `pack()` template to avoid this warning.

Attempt to use reference as lvalue in substr

(W substr) You supplied a reference as the first argument to `substr()` used as an lvalue, which is pretty strange. Perhaps you forgot to dereference it first. See [substr](#).

Bad arg length for %s, is %d, should be %s

(F) You passed a buffer of the wrong size to one of `msgctl()`, `semctl()` or `shmctl()`. In C parlance, the correct sizes are, respectively, `sizeof(struct msqid_ds *)`, `sizeof(struct semid_ds *)`, and `sizeof(struct shmid_ds *)`.

Bad evalled substitution pattern

(F) You've used the `/e` switch to evaluate the replacement for a substitution, but perl found a syntax error in the code to evaluate, most likely an unexpected right brace `}`.

Bad filehandle: %s

(F) A symbol was passed to something wanting a filehandle, but the symbol has no filehandle associated with it. Perhaps you didn't do an `open()`, or did it in another package.

Bad free() ignored

(S malloc) An internal routine called `free()` on something that had never been `malloc()`ed in the first place. Mandatory, but can be disabled by setting environment variable `PERL_BADFREETO` to 0.

This message can be seen quite often with `DB_File` on systems with "hard" dynamic linking, like AIX and OS/2. It is a bug of Berkeley DB which is left unnoticed if DB uses *forgiving* system `malloc()`.

Bad hash

(P) One of the internal hash routines was passed a null HV pointer.

Bad index while coercing array into hash

(F) The index looked up in the hash found as the 0'th element of a pseudo-hash is not legal. Index values must be at 1 or greater. See [perlref](#).

Badly placed () 's

(A) You've accidentally run your script through **csh** instead of Perl. Check the **#!** line, or manually feed your script into Perl yourself.

Bad name after %s::

(F) You started to name a symbol by using a package prefix, and then didn't finish the symbol. In particular, you can't interpolate outside of quotes, so

```
$var = 'myvar';
$sym = mypack::$var;
```

is not the same as

```
$var = 'myvar';
$sym = "mypack::$var";
```

Bad realloc() ignored

(S malloc) An internal routine called `realloc()` on something that had never been `malloc()`ed in the first place. Mandatory, but can be disabled by setting environment variable `PERL_BADFREE` to 1.

Bad symbol for array

(P) An internal request asked to add an array entry to something that wasn't a symbol table entry.

Bad symbol for filehandle

(P) An internal request asked to add a filehandle entry to something that wasn't a symbol table entry.

Bad symbol for hash

(P) An internal request asked to add a hash entry to something that wasn't a symbol table entry.

Bareword found in conditional

(W bareword) The compiler found a bareword where it expected a conditional, which often indicates that an `||` or `&&` was parsed as part of the last argument of the previous construct, for example:

```
open FOO || die;
```

It may also indicate a misspelled constant that has been interpreted as a bareword:

```
use constant TYPO => 1;
if (TYOP) { print "foo" }
```

The `strict pragma` is useful in avoiding such errors.

Bareword "%s" not allowed while "strict subs" in use

(F) With "strict subs" in use, a bareword is only allowed as a subroutine identifier, in curly brackets or to the left of the `=` symbol. Perhaps you need to predeclare a subroutine?

Bareword "%s" refers to nonexistent package

(W bareword) You used a qualified bareword of the form `FOO::`, but the compiler saw no other uses of that namespace before that point. Perhaps you need to predeclare a package?

BEGIN failed—compilation aborted

(F) An untrapped exception was raised while executing a `BEGIN` subroutine. Compilation stops immediately and the interpreter is exited.

BEGIN not safe after errors—compilation aborted

(F) Perl found a `BEGIN { }` subroutine (or a `use` directive, which implies a `BEGIN { }`) after one or more compilation errors had already occurred. Since the intended environment for the `BEGIN { }` could not be guaranteed (due to the errors), and since subsequent code likely depends on its correct operation, Perl just gave up.

`\1` better written as `$1`

(W syntax) Outside of patterns, backreferences live on as variables. The use of backslashes is grandfathered on the right-hand side of a substitution, but stylistically it's better to use the variable form because other Perl programmers will expect it, and it works better if there are more than 9 backreferences.

Binary number `0b11111111111111111111111111111111` non-portable

(W portable) The binary number you specified is larger than $2^{32}-1$ (4294967295) and therefore non-portable between systems. See [perlport](#) for more on portability concerns.

`bind()` on closed socket `%s`

(W closed) You tried to do a bind on a closed socket. Did you forget to check the return value of your `socket()` call? See [bind](#).

`binmode()` on closed filehandle `%s`

(W unopened) You tried `binmode()` on a filehandle that was never opened. Check you control flow and number of arguments.

Bit vector size 32 non-portable

(W portable) Using bit vector sizes larger than 32 is non-portable.

Bizarre copy of `%s` in `%s`

(P) Perl detected an attempt to copy an internal value that is not copyable.

-P not allowed for `setuid/setgid` script

(F) The script would have to be opened by the C preprocessor by name, which provides a race condition that breaks security.

Buffer overflow in `prime_env_iter: %s`

(W internal) A warning peculiar to VMS. While Perl was preparing to iterate over `%ENV`, it encountered a logical name or symbol definition which was too long, so it was truncated to the string shown.

Callback called `exit`

(F) A subroutine invoked from an external package via `call_sv()` exited by calling `exit`.

`%s()` called too early to check prototype

(W prototype) You've called a function that has a prototype before the parser saw a definition or declaration for it, and Perl could not check that the call conforms to the prototype. You need to either add an early prototype declaration for the subroutine in question, or move the subroutine definition ahead of the call to get proper prototype checking. Alternatively, if you are certain that you're calling the function correctly, you may put an ampersand before the name to avoid the warning. See [perlsyn](#).

`/` cannot take a count

(F) You had an `unpack` template indicating a counted-length string, but you have also specified an explicit size for the string. See [pack](#).

Can't bless non-reference value

(F) Only hard references may be blessed. This is how Perl "enforces" encapsulation of objects. See [perlobj](#).

Can't call method `"%s"` in empty package `"%s"`

(F) You called a method correctly, and it correctly indicated a package functioning as a class, but that package doesn't have ANYTHING defined in it, let alone methods. See [perlobj](#).

Can't call method `"%s"` on an undefined value

(F) You used the syntax of a method call, but the slot filled by the object reference or package name contains an undefined value. Something like this will reproduce the error:

```
$BADREF = undef;
process $BADREF 1,2,3;
$BADREF->process(1,2,3);
```

Can't call method "%s" on unblessed reference

(F) A method call must know in what package it's supposed to run. It ordinarily finds this out from the object reference you supply, but you didn't supply an object reference in this case. A reference isn't an object reference until it has been blessed. See [perlobj](#).

Can't call method "%s" without a package or object reference

(F) You used the syntax of a method call, but the slot filled by the object reference or package name contains an expression that returns a defined value which is neither an object reference nor a package name. Something like this will reproduce the error:

```
$BADREF = 42;
process $BADREF 1,2,3;
$BADREF->process(1,2,3);
```

Can't chdir to %s

(F) You called `perl -x/foo/bar`, but `/foo/bar` is not a directory that you can `chdir` to, possibly because it doesn't exist.

Can't check filesystem of script "%s" for nosuid

(P) For some reason you can't check the filesystem of the script for `nosuid`.

Can't coerce array into hash

(F) You used an array where a hash was expected, but the array has no information on how to map from keys to array indices. You can do that only with arrays that have a hash reference at index 0.

Can't coerce %s to integer in %s

(F) Certain types of SVs, in particular real symbol table entries (typeglobs), can't be forced to stop being what they are. So you can't say things like:

```
*foo += 1;
```

You CAN say

```
$foo = *foo;
$foo += 1;
```

but then `$foo` no longer contains a glob.

Can't coerce %s to number in %s

(F) Certain types of SVs, in particular real symbol table entries (typeglobs), can't be forced to stop being what they are.

Can't coerce %s to string in %s

(F) Certain types of SVs, in particular real symbol table entries (typeglobs), can't be forced to stop being what they are.

Can't create pipe mailbox

(P) An error peculiar to VMS. The process is suffering from exhausted quotas or other plumbing problems.

Can't declare class for non-scalar %s in "%s"

(F) Currently, only scalar variables can be declared with a specific class qualifier in a "my" or "our" declaration. The semantics may be extended for other types of variables in future.

Can't declare %s in "%s"

(F) Only scalar, array, and hash variables may be declared as "my" or "our" variables. They must have ordinary identifiers as names.

Can't do inplace edit: %s is not a regular file

(S inplace) You tried to use the `-i` switch on a special file, such as a file in `/dev`, or a FIFO. The file was ignored.

Can't do inplace edit on %s: %s

(S inplace) The creation of the new file failed for the indicated reason.

Can't do inplace edit without backup

(F) You're on a system such as MS-DOS that gets confused if you try reading from a deleted (but still opened) file. You have to say `-i .bak`, or some such.

Can't do inplace edit: %s would not be unique

(S inplace) Your filesystem does not support filenames longer than 14 characters and Perl was unable to create a unique filename during inplace editing with the `-i` switch. The file was ignored.

Can't do {n,m} with n m in regex; marked by <— HERE in m/%s/

(F) Minima must be less than or equal to maxima. If you really want your regexp to match something 0 times, just put `{0}`. The `<— HERE` shows in the regular expression about where the problem was discovered. See [perlre](#).

Can't do setegid!

(P) The `setegid()` call failed for some reason in the `setuid` emulator of `suidperl`.

Can't do seteuid!

(P) The `setuid` emulator of `suidperl` failed for some reason.

Can't do setuid

(F) This typically means that ordinary perl tried to exec `suidperl` to do `setuid` emulation, but couldn't exec it. It looks for a name of the form `sperl5.000` in the same directory that the perl executable resides under the name `perl5.000`, typically `/usr/local/bin` on Unix machines. If the file is there, check the execute permissions. If it isn't, ask your sysadmin why he and/or she removed it.

Can't do waitpid with flags

(F) This machine doesn't have either `waitpid()` or `wait4()`, so only `waitpid()` without flags is emulated.

Can't emulate -%s on #! line

(F) The `#!` line specifies a switch that doesn't make sense at this point. For example, it'd be kind of silly to put a `-x` on the `#!` line.

Can't exec "%s": %s

(W exec) A `system()`, `exec()`, or piped open call could not execute the named program for the indicated reason. Typical reasons include: the permissions were wrong on the file, the file wasn't found in `$ENV{PATH}`, the executable in question was compiled for another architecture, or the `#!` line in a script points to an interpreter that can't be run for similar reasons. (Or maybe your system doesn't support `#!` at all.)

Can't exec %s

(F) Perl was trying to execute the indicated program for you because that's what the `#!` line said. If that's not what you wanted, you may need to mention "perl" on the `#!` line somewhere.

Can't execute %s

(F) You used the `-S` switch, but the copies of the script to execute found in the `PATH` did not have correct permissions.

Can't find an opnumber for "%s"

(F) A string of a form `CORE::word` was given to `prototype()`, but there is no builtin with the name `word`.

Can't find %s character property "%s"

(F) You used `\p{ }` or `\P{ }` but the character property by that name could not be found. Maybe you misspelled the name of the property (remember that the names of character properties consist only of alphanumeric characters), or maybe you forgot the `Is` or `In` prefix?

Can't find label %s

(F) You said to `goto` a label that isn't mentioned anywhere that it's possible for us to go to. See [goto](#).

Can't find %s on PATH

(F) You used the `-S` switch, but the script to execute could not be found in the PATH.

Can't find %s on PATH, '.' not in PATH

(F) You used the `-S` switch, but the script to execute could not be found in the PATH, or at least not with the correct permissions. The script exists in the current directory, but PATH prohibits running it.

Can't find string terminator %s anywhere before EOF

(F) Perl strings can stretch over multiple lines. This message means that the closing delimiter was omitted. Because bracketed quotes count nesting levels, the following is missing its final parenthesis:

```
print q(The character '(' starts a side comment.);
```

If you're getting this error from a here-document, you may have included unseen whitespace before or after your closing tag. A good programmer's editor will have a way to help you find these characters.

Can't find %s property definition %s

(F) You may have tried to use `\p` which means a Unicode property (for example `\p{Lu}` is all uppercase letters). If you did mean to use a Unicode property, see [perlunicode](#) for the list of known properties. If you didn't mean to use a Unicode property, escape the `\p`, either by `\\p` (just the `\p`) or by `\Q\p` (the rest of the string, until possible `\E`).

Can't fork

(F) A fatal error occurred while trying to fork while opening a pipeline.

Can't get filespec – stale stat buffer?

(S) A warning peculiar to VMS. This arises because of the difference between access checks under VMS and under the Unix model Perl assumes. Under VMS, access checks are done by filename, rather than by bits in the stat buffer, so that ACLs and other protections can be taken into account. Unfortunately, Perl assumes that the stat buffer contains all the necessary information, and passes it, instead of the filespec, to the access checking routine. It will try to retrieve the filespec using the device name and FID present in the stat buffer, but this works only if you haven't made a subsequent call to the CRTL `stat()` routine, because the device name is overwritten with each call. If this warning appears, the name lookup failed, and the access checking routine gave up and returned FALSE, just to be conservative. (Note: The access checking routine knows about the Perl `stat` operator and file tests, so you shouldn't ever see this warning in response to a Perl command; it arises only if some internal code takes stat buffers lightly.)

Can't get pipe mailbox device name

(P) An error peculiar to VMS. After creating a mailbox to act as a pipe, Perl can't retrieve its name for later use.

Can't get SYSGEN parameter value for MAXBUF

(P) An error peculiar to VMS. Perl asked `$GETSYI` how big you want your mailbox buffers to be, and didn't get an answer.

Can't "goto" into the middle of a foreach loop

(F) A "goto" statement was executed to jump into the middle of a foreach loop. You can't get there from here. See [goto](#).

Can't "goto" out of a pseudo block

(F) A "goto" statement was executed to jump out of what might look like a block, except that it isn't a proper block. This usually occurs if you tried to jump out of a `sort()` block or subroutine, which is a no-no. See [goto](#).

Can't goto subroutine from an eval-string

(F) The "goto subroutine" call can't be used to jump out of an eval "string". (You can use it to jump out of an eval {BLOCK}, but you probably don't want to.)

Can't goto subroutine outside a subroutine

(F) The deeply magical "goto subroutine" call can only replace one subroutine call for another. It can't manufacture one out of whole cloth. In general you should be calling it out of only an AUTOLOAD routine anyway. See [goto](#).

Can't ignore signal CHLD, forcing to default

(W signal) Perl has detected that it is being run with the SIGCHLD signal (sometimes known as SIGCLD) disabled. Since disabling this signal will interfere with proper determination of exit status of child processes, Perl has reset the signal to its default value. This situation typically indicates that the parent program under which Perl may be running (e.g. cron) is being very careless.

Can't "last" outside a loop block

(F) A "last" statement was executed to break out of the current block, except that there's this itty bitty problem called there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block, as doesn't a block given to `sort()`, `map()` or `grep()`. You can usually double the curlyies to get the same effect though, because the inner curlyies will be considered a block that loops once. See [last](#).

Can't localize lexical variable %s

(F) You used `local` on a variable name that was previously declared as a lexical variable using "my". This is not allowed. If you want to localize a package variable of the same name, qualify it with the package name.

Can't localize pseudo-hash element

(F) You said something like `< local $ar-{ 'key' }`, where `$ar` is a reference to a pseudo-hash. That hasn't been implemented yet, but you can get a similar effect by localizing the corresponding array element directly — `< local $ar-[$ar-[0]{ 'key' }]`.

Can't localize through a reference

(F) You said something like `local $$ref`, which Perl can't currently handle, because when it goes to restore the old value of whatever `$ref` pointed to after the scope of the `local()` is finished, it can't be sure that `$ref` will still be a reference.

Can't locate %s

(F) You said to `do` (or `require`, or `use`) a file that couldn't be found. Perl looks for the file in all the locations mentioned in `@INC`, unless the file name included the full path to the file. Perhaps you need to set the `PERL5LIB` or `PERL5OPT` environment variable to say where the extra library is, or maybe the script needs to add the library name to `@INC`. Or maybe you just misspelled the name of the file. See [require](#) and [lib](#).

Can't locate auto/%s.al in @INC

(F) A function (or method) was called in a package which allows autoload, but there is no function to autoload. Most probable causes are a misprint in a function/method name or a failure to `AutoSplit` the file, say, by doing `make install`.

Can't locate object method "%s" via package "%s"

(F) You called a method correctly, and it correctly indicated a package functioning as a class, but that package doesn't define that particular method, nor does any of its base classes. See [perlobj](#).

Can't locate PerlIO%s

(F) You tried to use in `open()` a PerlIO layer that does not exist, e.g. `open(FH, ":nosuchlayer", "somefile")`.

(perhaps you forgot to load "%s"?)

(F) This is an educated guess made in conjunction with the message "Can't locate object method \"%s\" via package \"%s\"". It often means that a method requires a package that has not been loaded.

Can't locate package %s for @%s::ISA

(W syntax) The `@ISA` array contained the name of another package that doesn't seem to exist.

Can't make list assignment to \%ENV on this system

(F) List assignment to `%ENV` is not supported on some systems, notably VMS.

Can't modify %s in %s

(F) You aren't allowed to assign to the item indicated, or otherwise try to change it, such as with an auto-increment.

Can't modify nonexistent substring

(P) The internal routine that does assignment to a `substr()` was handed a `NULL`.

Can't modify non-lvalue subroutine call

(F) Subroutines meant to be used in lvalue context should be declared as such, see [Lvalue subroutines in perlsub](#).

Can't msgrcv to read-only var

(F) The target of a `msgrcv` must be modifiable to be used as a receive buffer.

Can't "next" outside a loop block

(F) A "next" statement was executed to reiterate the current block, but there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block, as doesn't a block given to `sort()`, `map()` or `grep()`. You can usually double the curlyes to get the same effect though, because the inner curlyes will be considered a block that loops once. See [next](#).

Can't open %s: %s

(S inplace) The implicit opening of a file through use of the `< < filehandle`, either implicitly under the `-n` or `-p` command-line switches, or explicitly, failed for the indicated reason. Usually this is because you don't have read permission for a file which you named on the command line.

Can't open a reference

(W io) You tried to open a scalar reference for reading or writing, using the 3-arg `open()` syntax :

```
open FH, '>', $ref;
```

but your version of perl is compiled without `perlio`, and this form of `open` is not supported.

Can't open bidirectional pipe

(W pipe) You tried to say `open(CMD, "|cmd|")`, which is not supported. You can try any of several modules in the Perl library to do this, such as `IPC::Open2`. Alternately, direct the pipe's output to a file using `">file"`, and then read it in under a different file handle.

Can't open error file %s as stderr

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after `'2'` or `'2'` on the command line for writing.

Can't open input file %s as stdin

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after '<' on the command line for reading.

Can't open output file %s as stdout

(F) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the file specified after '' or '' on the command line for writing.

Can't open output pipe (name: %s)

(P) An error peculiar to VMS. Perl does its own command line redirection, and couldn't open the pipe into which to send data destined for stdout.

Can't open perl script%s: %s

(F) The script you specified can't be opened for the indicated reason.

Can't read CRTL environ

(S) A warning peculiar to VMS. Perl tried to read an element of %ENV from the CRTL's internal environment array and discovered the array was missing. You need to figure out where your CRTL misplaced its environ or define *PERL_ENV_TABLES* (see [perlvm](#)) so that environ is not searched.

Can't redefine active sort subroutine %s

(F) Perl optimizes the internal handling of sort subroutines and keeps pointers into them. You tried to redefine one such sort subroutine when it was currently active, which is not allowed. If you really want to do this, you should write `sort { &func } @x` instead of `sort func @x`.

Can't "redo" outside a loop block

(F) A "redo" statement was executed to restart the current block, but there isn't a current block. Note that an "if" or "else" block doesn't count as a "loopish" block, as doesn't a block given to `sort()`, `map()` or `grep()`. You can usually double the curlyes to get the same effect though, because the inner curlyes will be considered a block that loops once. See [redo](#).

Can't remove %s: %s, skipping file

(S inplace) You requested an inplace edit without creating a backup file. Perl was unable to remove the original file to replace it with the modified file. The file was left unmodified.

Can't rename %s to %s: %s, skipping file

(S inplace) The rename done by the `-i` switch failed for some reason, probably because you don't have write permission to the directory.

Can't reopen input pipe (name: %s) in binary mode

(P) An error peculiar to VMS. Perl thought stdin was a pipe, and tried to reopen it to accept binary data. Alas, it failed.

Can't resolve method '%s' overloading '%s' in package '%s'

(F|P) Error resolving overloading specified by a method name (as opposed to a subroutine reference): no such method callable via the package. If method name is ???, this is an internal error.

Can't reswap uid and euid

(P) The `setreuid()` call failed for some reason in the `setuid` emulator of `suidperl`.

Can't return %s from lvalue subroutine

(F) Perl detected an attempt to return illegal lvalues (such as temporary or readonly values) from a subroutine used as an lvalue. This is not allowed.

Can't return %s to lvalue scalar context

(F) You tried to return a complete array or hash from an lvalue subroutine, but you called the subroutine in a way that made Perl think you meant to return only one value. You probably meant to write parentheses around the call to the subroutine, which tell Perl that the call should be in list

context.

Can't return outside a subroutine

(F) The return statement was executed in mainline code, that is, where there was no subroutine call to return out of. See [perlsub](#).

Can't stat script "%s"

(P) For some reason you can't `fstat()` the script even though you have it open already. Bizarre.

Can't swap uid and euid

(P) The `setreuid()` call failed for some reason in the `setuid` emulator of `suidperl`.

Can't take log of %g

(F) For ordinary real numbers, you can't take the logarithm of a negative number or zero. There's a `Math::Complex` package that comes standard with Perl, though, if you really want to do that for the negative numbers.

Can't take sqrt of %g

(F) For ordinary real numbers, you can't take the square root of a negative number. There's a `Math::Complex` package that comes standard with Perl, though, if you really want to do that.

Can't undef active subroutine

(F) You can't undefine a routine that's currently running. You can, however, redefine it while it's running, and you can even undef the redefined subroutine while the old routine is running. Go figure.

Can't unshift

(F) You tried to unshift an "unreal" array that can't be unshifted, such as the main Perl stack.

Can't upgrade that kind of scalar

(P) The internal `sv_upgrade` routine adds "members" to an SV, making it into a more specialized kind of SV. The top several SV types are so specialized, however, that they cannot be interconverted. This message indicates that such a conversion was attempted.

Can't upgrade to undef

(P) The undefined SV is the bottom of the totem pole, in the scheme of upgradability. Upgrading to undef indicates an error in the code calling `sv_upgrade`.

Can't use an undefined value as %s reference

(F) A value used as either a hard reference or a symbolic reference must be a defined value. This helps to delurk some insidious errors.

Can't use anonymous symbol table for method lookup

(P) The internal routine that does method lookup was handed a symbol table that doesn't have a name. Symbol tables can become anonymous for example by undefining stashes: `undef %Some::Package::`.

Can't use bareword ("%s") as %s ref while "strict refs" in use

(F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See [perlref](#).

Can't use %! because Errno.pm is not available

(F) The first time the %! hash is used, perl automatically loads the `Errno.pm` module. The `Errno` module is expected to tie the %! hash to provide symbolic names for \$! `errno` values.

Can't use %s for loop variable

(F) Only a simple scalar variable may be used as a loop variable on a `foreach`.

Can't use global %s in "my"

(F) You tried to declare a magical variable as a lexical variable. This is not allowed, because the magic can be tied to only one location (namely the global variable) and it would be incredibly confusing to have variables in your program that looked like magical variables but weren't.

Can't use "my %s" in sort comparison

(F) The global variables \$a and \$b are reserved for sort comparisons. You mentioned \$a or \$b in the same line as the <= or cmp operator, and the variable had earlier been declared as a lexical variable. Either qualify the sort variable with the package name, or rename the lexical variable.

Can't use %s ref as %s ref

(F) You've mixed up your reference types. You have to dereference a reference of the type needed. You can use the `ref()` function to test the type of the reference, if need be.

Can't use string ("%s") as %s ref while "strict refs" in use

(F) Only hard references are allowed by "strict refs". Symbolic references are disallowed. See [perlref](#).

Can't use subscript on %s

(F) The compiler tried to interpret a bracketed expression as a subscript. But to the left of the brackets was an expression that didn't look like an array reference, or anything else subscriptable.

Can't use \%c to mean \$%c in expression

(W syntax) In an ordinary expression, backslash is a unary operator that creates a reference to its argument. The use of backslash to indicate a backreference to a matched substring is valid only as part of a regular expression pattern. Trying to do this in ordinary Perl code produces a value that prints out looking like `SCALAR(0xdecaf)`. Use the `$1` form instead.

Can't weaken a nonreference

(F) You attempted to weaken something that was not a reference. Only references can be weakened.

Can't x= to read-only value

(F) You tried to repeat a constant value (often the undefined value) with an assignment operator, which implies modifying the value itself. Perhaps you need to copy the value to a temporary, and repeat that.

Character in "C" format wrapped

(W pack) You said

```
pack("C", $x)
```

where \$x is either less than 0 or more than 255; the "C" format is only for encoding native operating system characters (ASCII, EBCDIC, and so on) and not for Unicode characters, so Perl behaved as if you meant

```
pack("C", $x & 255)
```

If you actually want to pack Unicode codepoints, use the "U" format instead.

Character in "c" format wrapped

(W pack) You said

```
pack("c", $x)
```

where \$x is either less than -128 or more than 127; the "c" format is only for encoding native operating system characters (ASCII, EBCDIC, and so on) and not for Unicode characters, so Perl behaved as if you meant

```
pack("c", $x & 255);
```

If you actually want to pack Unicode codepoints, use the "U" format instead.

close() on unopened filehandle %s

(W unopened) You tried to close a filehandle that was never opened.

%s: Command not found

(A) You've accidentally run your script through **cs**h instead of Perl. Check the **#!** line, or manually feed your script into Perl yourself.

Compilation failed in require

(F) Perl could not compile a file specified in a `require` statement. Perl uses this generic message when none of the errors that it encountered were severe enough to halt compilation immediately.

Complex regular subexpression recursion limit (%d) exceeded

(W regexp) The regular expression engine uses recursion in complex situations where back-tracking is required. Recursion depth is limited to 32766, or perhaps less in architectures where the stack cannot grow arbitrarily. ("Simple" and "medium" situations are handled without recursion and are not subject to a limit.) Try shortening the string under examination; looping in Perl code (e.g. with `while`) rather than in the regular expression engine; or rewriting the regular expression so that it is simpler or backtracks less. (See [perlfaq2](#) for information on *Mastering Regular Expressions*.)

cond_broadcast() called on unlocked variable

(W threads) Within a thread-enabled program, you tried to call `cond_broadcast()` on a variable which wasn't locked. The `cond_broadcast()` function is used to wake up another thread that is waiting in a `cond_wait()`. To ensure that the signal isn't sent before the other thread has a chance to enter the wait, it is usual for the signaling thread to first wait for a lock on variable. This lock attempt will only succeed after the other thread has entered `cond_wait()` and thus relinquished the lock.

cond_signal() called on unlocked variable

(W threads) Within a thread-enabled program, you tried to call `cond_signal()` on a variable which wasn't locked. The `cond_signal()` function is used to wake up another thread that is waiting in a `cond_wait()`. To ensure that the signal isn't sent before the other thread has a chance to enter the wait, it is usual for the signaling thread to first wait for a lock on variable. This lock attempt will only succeed after the other thread has entered `cond_wait()` and thus relinquished the lock.

connect() on closed socket %s

(W closed) You tried to do a connect on a closed socket. Did you forget to check the return value of your `socket()` call? See [connect](#).

Constant(%s)%s: %s

(F) The parser found inconsistencies either while attempting to define an overloaded constant, or when trying to find the character name specified in the `\N{...}` escape. Perhaps you forgot to load the corresponding overload or `chardnames` pragma? See [chardnames](#) and [overload](#).

Constant is not %s reference

(F) A constant value (perhaps declared using the `use constant` pragma) is being dereferenced, but it amounts to the wrong type of reference. The message indicates the type of reference that was expected. This usually indicates a syntax error in dereferencing the constant value. See [Constant Functions in perlsub](#) and [constant](#).

Constant subroutine %s redefined

(S) You redefined a subroutine which had previously been eligible for inlining. See [Constant Functions in perlsub](#) for commentary and workarounds.

Constant subroutine %s undefined

(W misc) You undefined a subroutine which had previously been eligible for inlining. See [Constant Functions in perlsub](#) for commentary and workarounds.

Copy method did not return a reference

(F) The method which overloads "=" is buggy. See [Copy Constructor](#).

CORE::%s is not a keyword

(F) The CORE:: namespace is reserved for Perl keywords.

corrupted regexp pointers

(P) The regular expression engine got confused by what the regular expression compiler gave it.

corrupted regexp program

(P) The regular expression engine got passed a regexp program without a valid magic number.

Corrupt malloc ptr 0x%x at 0x%x

(P) The malloc package that comes with Perl had an internal failure.

-p destination: %s

(F) An error occurred during the implicit output invoked by the -p command-line switch. (This output goes to STDOUT unless you've redirected it with `select()`.)

-T and -B not implemented on filehandles

(F) Perl can't peek at the stdio buffer of filehandles when it doesn't know about your kind of stdio. You'll have to use a filename instead.

Deep recursion on subroutine "%s"

(W recursion) This subroutine has called itself (directly or indirectly) 100 times more than it has returned. This probably indicates an infinite recursion, unless you're writing strange benchmark programs, in which case it indicates something else.

defined(@array) is deprecated

(D deprecated) `defined()` is not usually useful on arrays because it checks for an undefined *scalar* value. If you want to see if the array is empty, just use `if (@array) { # not empty }` for example.

defined(%hash) is deprecated

(D deprecated) `defined()` is not usually useful on hashes because it checks for an undefined *scalar* value. If you want to see if the hash is empty, just use `if (%hash) { # not empty }` for example.

%s defines neither package nor VERSION—version check failed

(F) You said something like "use Module 42" but in the Module file there are neither package declarations nor a \$VERSION.

Delimiter for here document is too long

(F) In a here document construct like `<<FOO`, the label FOO is too long for Perl to handle. You have to be seriously twisted to write code that triggers this error.

Did not produce a valid header

See Server error.

%s did not return a true value

(F) A required (or used) file must return a true value to indicate that it compiled correctly and ran its initialization code correctly. It's traditional to end such a file with a "1;", though any true value would do. See [require](#).

(Did you mean &%s instead?)

(W) You probably referred to an imported subroutine &FOO as \$FOO or some such.

(Did you mean "local" instead of "our"?)

(W misc) Remember that "our" does not localize the declared global variable. You have declared it again in the same lexical scope, which seems superfluous.

(Did you mean \$ or @ instead of %?)

(W) You probably said `%hash{$key}` when you meant `$hash{$key}` or `@hash{@keys}`. On the other hand, maybe you just meant `%hash` and got carried away.

Died

(F) You passed `die()` an empty string (the equivalent of `die ""`) or you called it with no args and both `$@` and `$_` were empty.

Document contains no data

See Server error.

%s does not define %s::VERSION—version check failed

(F) You said something like "use Module 42" but the Module did not define a `$VERSION`.

Don't know how to handle magic of type '%s'

(P) The internal handling of magical variables has been cursed.

do_study: out of memory

(P) This should have been caught by `safemalloc()` instead.

(Do you need to predeclare %s?)

(S) This is an educated guess made in conjunction with the message "%s found where operator expected". It often means a subroutine or module name is being referenced that hasn't been declared yet. This may be because of ordering problems in your file, or because of a missing "sub", "package", "require", or "use" statement. If you're referencing something that isn't defined yet, you don't actually have to define the subroutine or package before the current location. You can use an empty "sub foo;" or "package FOO;" to enter a "forward" declaration.

`dump()` better written as `CORE::dump()`

(W misc) You used the obsolescent `dump()` built-in function, without fully qualifying it as `CORE::dump()`. Maybe it's a typo. See [dump](#).

Duplicate `free()` ignored

(S malloc) An internal routine called `free()` on something that had already been freed.

elseif should be elsif

(S) There is no keyword "elseif" in Perl because Larry thinks it's ugly. Your code will be interpreted as an attempt to call a method named "elseif" for the class returned by the following block. This is unlikely to be what you want.

Empty %s

(F) `\p` and `\P` are used to introduce a named Unicode property, as described in [perlunicode](#) and [perltre](#). You used `\p` or `\P` in a regular expression without specifying the property name.

entering effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

Error converting file specification %s

(F) An error peculiar to VMS. Because Perl may have to deal with file specifications in either VMS or Unix syntax, it converts them to a single form when it must operate on them directly. Either you've passed an invalid file specification to Perl, or you've found a case the conversion routines don't handle. Drat.

%s: Eval-group in insecure regular expression

(F) Perl detected tainted data when trying to compile a regular expression that contains the `(?{ ... })` zero-width assertion, which is unsafe. See [\(?{ code }\)](#), and [perlsec](#).

%s: Eval-group not allowed at run time

(F) Perl tried to compile a regular expression containing the `(?{ ... })` zero-width assertion at run time, as it would when the pattern contains interpolated values. Since that is a security risk, it is not allowed. If you insist, you may still do this by explicitly building the pattern from an interpolated string at run time and using that in an `eval()`. See [\(?{ code }\)](#).

%s: Eval-group not allowed, use re 'eval'

(F) A regular expression contained the `(?{ ... })` zero-width assertion, but that construct is only allowed when the `use re 'eval'` pragma is in effect. See [\(?{ code }\)](#).

Excessively long < operator

(F) The contents of a `<` operator may not exceed the maximum size of a Perl identifier. If you're just trying to glob a long list of filenames, try using the `glob()` operator, or put the filenames into a variable and glob that.

exec? I'm not *that* kind of operating system

(F) The `exec` function is not implemented in MacPerl. See [perlport](#).

Execution of %s aborted due to compilation errors

(F) The final summary message when a Perl compilation fails.

Exiting eval via %s

(W exiting) You are exiting an eval by unconventional means, such as a `goto`, or a loop control statement.

Exiting format via %s

(W exiting) You are exiting a format by unconventional means, such as a `goto`, or a loop control statement.

Exiting pseudo-block via %s

(W exiting) You are exiting a rather special block construct (like a `sort` block or subroutine) by unconventional means, such as a `goto`, or a loop control statement. See [sort](#).

Exiting subroutine via %s

(W exiting) You are exiting a subroutine by unconventional means, such as a `goto`, or a loop control statement.

Exiting substitution via %s

(W exiting) You are exiting a substitution by unconventional means, such as a `return`, a `goto`, or a loop control statement.

Explicit blessing to "" (assuming package main)

(W misc) You are blessing a reference to a zero length string. This has the effect of blessing the reference into the package `main`. This is usually not what you want. Consider providing a default target package, e.g. `bless($ref, $p || 'MyPackage')`;

%s: Expression syntax

(A) You've accidentally run your script through `csh` instead of Perl. Check the `#!` line, or manually feed your script into Perl yourself.

%s failed—call queue aborted

(F) An untrapped exception was raised while executing a `CHECK`, `INIT`, or `END` subroutine. Processing of the remainder of the queue of such routines has been prematurely ended.

False [] range "%s" in regex; marked by <— HERE in m/%s/

(W regexp) A character class range must start and end at a literal character, not another character class like `\d` or `[:alpha:]`. The `"—"` in your false range is interpreted as a literal `"—"`. Consider quoting the `"—"`, `"\—"`. The `<— HERE` shows in the regular expression about where the problem was discovered. See [perlre](#).

Fatal VMS error at %s, line %d

(P) An error peculiar to VMS. Something untoward happened in a VMS system service or RTL routine; Perl's exit status should provide more details. The filename in "at %s" and the line number in "line %d" tell you which section of the Perl source code is distressed.

fcntl is not implemented

(F) Your machine apparently doesn't implement `fcntl()`. What is this, a PDP-11 or something?

Filehandle %s opened only for input

(W io) You tried to write on a read-only filehandle. If you intended it to be a read-write filehandle, you needed to open it with `"<+"` or `">+"` or `">+"` instead of with `"<+"` or nothing. If you intended only to write the file, use `">+"` or `">+"`. See [open](#). The warning will also occur if `STDOUT` (file descriptor 1) or `STDERR` (file descriptor 2) is opened for input, this is a pre-emptive warning in case some other part of your program or a child process is expecting `STDOUT` and `STDERR` to be writable. This can happen accidentally if you `close(STDOUT)` or `STDERR` and then `open` an unrelated handle which will reuse the lowest numbered available descriptor.

Filehandle %s opened only for output

(W io) You tried to read from a filehandle opened only for writing. If you intended it to be a read/write filehandle, you needed to open it with `"<+"` or `">+"` or `">+"` instead of with `"<+"` or nothing. If you intended only to read from the file, use `"<+"`. See [open](#). The warning will also occur if `STDIN` (file descriptor 0) is opened for output – this is a pre-emptive warning in case some other part of your program or a child process is expecting `STDIN` to be readable. This can happen accidentally if you `close(STDIN)` and then `open` an unrelated handle which will reuse the lowest numbered available descriptor.

Final \$ should be \\$ or \$name

(F) You must now decide whether the final `$` in a string was meant to be a literal dollar sign, or was meant to introduce a variable name that happens to be missing. So you have to put either the backslash or the name.

Final @ should be \@ or @name

(F) You must now decide whether the final `@` in a string was meant to be a literal `"@"` sign, or was meant to introduce a variable name that happens to be missing. So you have to put either the backslash or the name.

flock() on closed filehandle %s

(W closed) The filehandle you're attempting to `flock()` got itself closed some time before now. Check your control flow. `flock()` operates on filehandles. Are you attempting to call `flock()` on a dirhandle by the same name?

Quantifier follows nothing in regex;

marked by <— HERE in m/%s/

(F) You started a regular expression with a quantifier. Backslash it if you meant it literally. The `<— HERE` shows in the regular expression about where the problem was discovered. See [perlre](#).

Format not terminated

(F) A format must be terminated by a line with a solitary dot. Perl got to the end of your file without finding such a line.

Format %s redefined

(W redefine) You redefined a format. To suppress this warning, say

```
{
    no warnings 'redefine';
    eval "format NAME = ...";
}
```

Found = in conditional, should be ==

(W syntax) You said

```
if ($foo = 123)
```

when you meant

```
if ($foo == 123)
```

(or something like that).

%s found where operator expected

(S) The Perl lexer knows whether to expect a term or an operator. If it sees what it knows to be a term when it was expecting to see an operator, it gives you this warning. Usually it indicates that an operator or delimiter was omitted, such as a semicolon.

gdbm store returned %d, errno %d, key "%s"

(S) A warning from the GDBM_File extension that a store failed.

gethostent not implemented

(F) Your C library apparently doesn't implement `gethostent()`, probably because if it did, it'd feel morally obligated to return every hostname on the Internet.

get%lname() on closed socket %s

(W closed) You tried to get a socket or peer socket name on a closed socket. Did you forget to check the return value of your `socket()` call?

getpwnam returned invalid UIC %#o for user "%s"

(S) A warning peculiar to VMS. The call to `sys$getuai` underlying the `getpwnam` operator returned an invalid UIC.

getsockopt() on closed socket %s

(W closed) You tried to get a socket option on a closed socket. Did you forget to check the return value of your `socket()` call? See [getsockopt](#).

Global symbol "%s" requires explicit package name

(F) You've said "use strict vars", which indicates that all variables must either be lexically scoped (using "my"), declared beforehand using "our", or explicitly qualified to say which package the global variable is in (using "::").

glob failed (%s)

(W glob) Something went wrong with the external program(s) used for `glob` and `< <*.c`. Usually, this means that you supplied a `glob` pattern that caused the external program to fail and exit with a nonzero status. If the message indicates that the abnormal exit resulted in a coredump, this may also mean that your `csh` (C shell) is broken. If so, you should change all of the `csh`-related variables in `config.sh`: If you have `tcsh`, make the variables refer to it as if it were `csh` (e.g. `full_csh='/usr/bin/tcsh'`); otherwise, make them all empty (except that `d_csh` should be `'undef'`) so that Perl will think `csh` is missing. In either case, after editing `config.sh`, run `./Configure -S` and rebuild Perl.

Glob not terminated

(F) The lexer saw a left angle bracket in a place where it was expecting a term, so it's looking for the corresponding right angle bracket, and not finding it. Chances are you left some needed parentheses out earlier in the line, and you really meant a "less than".

Got an error from DosAllocMem

(P) An error peculiar to OS/2. Most probably you're using an obsolete version of Perl, and this should not happen anyway.

goto must have label

(F) Unlike with "next" or "last", you're not allowed to goto an unspecified destination. See [goto](#).

%s-group starts with a count

(F) In pack/unpack a ()-group started with a count. A count is supposed to follow something: a template character or a ()-group.

%s had compilation errors

(F) The final summary message when a perl -c fails.

Had to create %s unexpectedly

(S internal) A routine asked for a symbol from a symbol table that ought to have existed already, but for some reason it didn't, and had to be created on an emergency basis to prevent a core dump.

Hash %s missing the % in argument %d of %s()

(D deprecated) Really old Perl let you omit the % on hash names in some spots. This is now heavily deprecated.

%s has too many errors

(F) The parser has given up trying to parse the program after 10 errors. Further error messages would likely be uninformative.

Hexadecimal number 0xffffffff non-portable

(W portable) The hexadecimal number you specified is larger than $2^{32}-1$ (4294967295) and therefore non-portable between systems. See [perlport](#) for more on portability concerns.

Identifier too long

(F) Perl limits identifiers (names for variables, functions, etc.) to about 250 characters for simple names, and somewhat more for compound names (like \$A:B). You've exceeded Perl's limits. Future versions of Perl are likely to eliminate these arbitrary limitations.

Illegal binary digit %s

(F) You used a digit other than 0 or 1 in a binary number.

Illegal binary digit %s ignored

(W digit) You may have tried to use a digit other than 0 or 1 in a binary number. Interpretation of the binary number stopped before the offending digit.

Illegal character %s (carriage return)

(F) Perl normally treats carriage returns in the program text as it would any other whitespace, which means you should never see this error when Perl was built using standard options. For some reason, your version of Perl appears to have been built without this support. Talk to your Perl administrator.

Illegal character in prototype for %s : %s

(W syntax) An illegal character was found in a prototype declaration. Legal characters in prototypes are \$, @, %, *, :, [,], &, and \.

Illegal division by zero

(F) You tried to divide a number by 0. Either something was wrong in your logic, or you need to put a conditional in to guard against meaningless input.

Illegal hexadecimal digit %s ignored

(W digit) You may have tried to use a character other than 0 – 9 or A – F, a – f in a hexadecimal number. Interpretation of the hexadecimal number stopped before the illegal character.

Illegal modulus zero

(F) You tried to divide a number by 0 to get the remainder. Most numbers don't take to this kindly.

Illegal number of bits in vec

(F) The number of bits in `vec()` (the third argument) must be a power of two from 1 to 32 (or 64, if your platform supports that).

Illegal octal digit %s

(F) You used an 8 or 9 in an octal number.

Illegal octal digit %s ignored

(W digit) You may have tried to use an 8 or 9 in an octal number. Interpretation of the octal number stopped before the 8 or 9.

Illegal switch in PERL5OPT: %s

(X) The PERL5OPT environment variable may only be used to set the following switches: **–[DIMUdmtw]**.

Ill-formed CRTL environ value "%s"

(W internal) A warning peculiar to VMS. Perl tried to read the CRTL's internal environ array, and encountered an element without the = delimiter used to separate keys from values. The element is ignored.

Ill-formed message in prime_env_iter: |%s|

(W internal) A warning peculiar to VMS. Perl tried to read a logical name or CLI symbol definition when preparing to iterate over %ENV, and didn't see the expected delimiter between key and value, so the line was ignored.

(in cleanup) %s

(W misc) This prefix usually indicates that a `DESTROY()` method raised the indicated exception. Since destructors are usually called by the system at arbitrary points during execution, and often a vast number of times, the warning is issued only once for any number of failures that would otherwise result in the same message being repeated.

Failure of user callbacks dispatched using the `G_KEEPPERR` flag could also result in this warning. See [G_KEEPPERR](#).

In EBCDIC the v-string components cannot exceed 2147483647

(F) An error peculiar to EBCDIC. Internally, v-strings are stored as Unicode code points, and encoded in EBCDIC as UTF-EBCDIC. The UTF-EBCDIC encoding is limited to code points no larger than 2147483647 (0x7FFFFFFF).

Insecure dependency in %s

(F) You tried to do something that the tainting mechanism didn't like. The tainting mechanism is turned on when you're running `setuid` or `setgid`, or when you specify `-T` to turn it on explicitly. The tainting mechanism labels all data that's derived directly or indirectly from the user, who is considered to be unworthy of your trust. If any such data is used in a "dangerous" operation, you get this error. See [perlsec](#) for more information.

Insecure directory in %s

(F) You can't use `system()`, `exec()`, or a piped open in a `setuid` or `setgid` script if `$ENV{PATH}` contains a directory that is writable by the world. See [perlsec](#).

Insecure `$ENV{ %s }` while running %s

(F) You can't use `system()`, `exec()`, or a piped open in a `setuid` or `setgid` script if any of `$ENV{PATH}`, `$ENV{IFS}`, `$ENV{CDPATH}`, `$ENV{ENV}` or `$ENV{BASH_ENV}` are derived from data supplied (or potentially supplied) by the user. The script must set the path to a known value, using trustworthy data. See [perlsec](#).

Integer overflow in %s number

(W overflow) The hexadecimal, octal or binary number you have specified either as a literal or as an argument to `hex()` or `oct()` is too big for your architecture, and has been converted to a floating point number. On a 32-bit architecture the largest hexadecimal, octal or binary number representable without overflow is `0xFFFFFFFF`, `037777777777`, or `0b11111111111111111111111111111111` respectively. Note that Perl transparently promotes all numbers to a floating point representation internally—subject to loss of precision errors in subsequent operations.

Internal disaster in regex; marked by <— HERE in m/%s/

(P) Something went badly wrong in the regular expression parser. The <— HERE shows in the regular expression about where the problem was discovered.

Internal inconsistency in tracking vforks

(S) A warning peculiar to VMS. Perl keeps track of the number of times you've called `fork` and `exec`, to determine whether the current call to `exec` should affect the current script or a subprocess (see [exec LIST in perlvms](#)). Somehow, this count has become scrambled, so Perl is making a guess and treating this `exec` as a request to terminate the Perl script and execute the specified command.

Internal urp in regex; marked by <— HERE in m/%s/

(P) Something went badly awry in the regular expression parser. The <— HERE shows in the regular expression about where the problem was discovered.

%s (...) interpreted as function

(W syntax) You've run afoul of the rule that says that any list operator followed by parentheses turns into a function, with all the list operators arguments found inside the parentheses. See [Terms and List Operators \(Leftward\)](#).

Invalid %s attribute: %s

The indicated attribute for a subroutine or variable was not recognized by Perl or by a user-supplied handler. See [attributes](#).

Invalid %s attributes: %s

The indicated attributes for a subroutine or variable were not recognized by Perl or by a user-supplied handler. See [attributes](#).

Invalid conversion in %s: "%s"

(W printf) Perl does not understand the given format conversion. See [sprintf](#).

Invalid [] range "%s" in regex; marked by <— HERE in m/%s/

(F) The range specified in a character class had a minimum character greater than the maximum character. One possibility is that you forgot the `{ }` from your ending `\x{ } - \x` without the curly braces can go only up to `ff`. The <— HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

Invalid [] range "%s" in transliteration operator

(F) The range specified in the `tr///` or `y///` operator had a minimum character greater than the maximum character. See [perlop](#).

Invalid separator character %s in attribute list

(F) Something other than a colon or whitespace was seen between the elements of an attribute list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon. See [attributes](#).

Invalid type in pack: '%s'

(F) The given character is not a valid pack type. See [pack](#). (W pack) The given character is not a valid pack type but used to be silently ignored.

Invalid type in unpack: '%s'

(F) The given character is not a valid unpack type. See [unpack](#). (W unpack) The given character is not a valid unpack type but used to be silently ignored.

ioctl is not implemented

(F) Your machine apparently doesn't implement `ioctl()`, which is pretty strange for a machine that supports C.

ioctl() on unopened %s

(W unopened) You tried `ioctl()` on a filehandle that was never opened. Check you control flow and number of arguments.

IO::Socket::atmark not implemented on this architecture

(F) Your machine doesn't implement the `socketatmark()` functionality, neither as a system call or an `ioctl` call (SIOCATMARK).

'%s' is not a code reference

(W overload) The second (fourth, sixth, ...) argument of `overload::constant` needs to be a code reference. Either an anonymous subroutine, or a reference to a subroutine.

'%s' is not an overloadable type

(W overload) You tried to overload a constant type the overload package is unaware of.

junk on end of regexp

(P) The regular expression parser is confused.

Label not found for "last %s"

(F) You named a loop to break out of, but you're not currently in a loop of that name, not even if you count where you were called from. See [last](#).

Label not found for "next %s"

(F) You named a loop to continue, but you're not currently in a loop of that name, not even if you count where you were called from. See [last](#).

Label not found for "redo %s"

(F) You named a loop to restart, but you're not currently in a loop of that name, not even if you count where you were called from. See [last](#).

leaving effective %s failed

(F) While under the `use filetest` pragma, switching the real and effective uids or gids failed.

listen() on closed socket %s

(W closed) You tried to do a `listen` on a closed socket. Did you forget to check the return value of your `socket()` call? See [listen](#).

lstat() on filehandle %s

(W io) You tried to do an `lstat` on a filehandle. What did you mean by that? `lstat()` makes sense only on filenames. (Perl did a `fstat()` instead on the filehandle.)

Lvalue subs returning %s not implemented yet

(F) Due to limitations in the current implementation, array and hash values cannot be returned in subroutines used in lvalue context. See [Lvalue subroutines in perlsub](#).

Lookbehind longer than %d not implemented in regex;

marked by <— HERE in m/%s/

(F) There is currently a limit on the length of string which lookbehind can handle. This restriction may be eased in a future release. The <— HERE shows in the regular expression about where the problem was discovered.

Malformed PERLLIB_PREFIX

(F) An error peculiar to OS/2. PERLLIB_PREFIX should be of the form

```
prefix1;prefix2
```

or

```
prefix1 prefix2
```

with nonempty prefix1 and prefix2. If prefix1 is indeed a prefix of a builtin library search path, prefix2 is substituted. The error may appear if components are not found, or are too long. See "PERLLIB_PREFIX" in [perlos2](#).

Malformed prototype for %s: %s

(F) You tried to use a function with a malformed prototype. The syntax of function prototypes is given a brief compile-time check for obvious errors like invalid characters. A more rigorous check is run when the function is called.

Malformed UTF-8 character (%s)

Perl detected something that didn't comply with UTF-8 encoding rules.

One possible cause is that you read in data that you thought to be in UTF-8 but it wasn't (it was for example legacy 8-bit data). Another possibility is careless use of `utf8::upgrade()`.

Malformed UTF-16 surrogate

Perl thought it was reading UTF-16 encoded character data but while doing it Perl met a malformed Unicode surrogate.

%s matches null string many times in regex;

marked by <— HERE in m/%s/

(W regexp) The pattern you've specified would be an infinite loop if the regular expression engine didn't specifically check for that. The <— HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

"%s" may clash with future reserved word

(W) This warning may be due to running a perl5 script through a perl4 interpreter, especially if the word that is being warned about is "use" or "my".

% may only be used in unpack

(F) You can't pack a string by supplying a checksum, because the checksumming process loses information, and you can't go the other way. See [unpack](#).

Method for operation %s not found in package %s during blessing

(F) An attempt was made to specify an entry in an overloading table that doesn't resolve to a valid subroutine. See [overload](#).

Method %s not permitted

See Server error.

Might be a runaway multi-line %s string starting on line %d

(S) An advisory indicating that the previous error may have been caused by a missing delimiter on a string or pattern, because it eventually ended earlier on the current line.

Misplaced _ in number

(W syntax) An underscore (underbar) in a numeric constant did not separate two digits.

Missing %sbrace%s on \N{}

(F) Wrong syntax of character name literal \N{charname} within double-quotish context.

Missing comma after first argument to %s function

(F) While certain functions allow you to specify a filehandle or an "indirect object" before the argument list, this ain't one of them.

Missing command in piped open

(W pipe) You used the `open(FH, " | command")` or `open(FH, "command |")` construction, but the command was missing or blank.

Missing name in "my sub"

(F) The reserved syntax for lexically scoped subroutines requires that they have a name with which they can be found.

Missing \$ on loop variable

(F) Apparently you've been programming in **cs**h too much. Variables are always mentioned with the \$ in Perl, unlike in the shells, where it can vary from one line to the next.

(Missing operator before %s?)

(S) This is an educated guess made in conjunction with the message "%s found where operator expected". Often the missing operator is a comma.

Missing right brace on %s

(F) Missing right brace in `\p{...}` or `\P{...}`.

Missing right curly or square bracket

(F) The lexer counted more opening curly or square brackets than closing ones. As a general rule, you'll find it's missing near the place you were last editing.

(Missing semicolon on previous line?)

(S) This is an educated guess made in conjunction with the message "%s found where operator expected". Don't automatically put a semicolon on the previous line just because you saw this message.

Modification of a read-only value attempted

(F) You tried, directly or indirectly, to change the value of a constant. You didn't, of course, try `"2 = 1"`, because the compiler catches that. But an easy way to do the same thing is:

```
sub mod { $_[0] = 1 }
mod(2);
```

Another way is to assign to a `substr()` that's off the end of the string.

Yet another way is to assign to a `foreach` loop *VAR* when *VAR* is aliased to a constant in the look *LIST*:

```
$x = 1;
foreach my $n ($x, 2) {
    $n *= 2; # modifies the $x, but fails on attempt to modify the 2
}
```

Modification of non-creatable array value attempted, %s

(F) You tried to make an array value spring into existence, and the subscript was probably negative, even counting from end of the array backwards.

Modification of non-creatable hash value attempted, %s

(P) You tried to make a hash value spring into existence, and it couldn't be created for some peculiar reason.

Module name must be constant

(F) Only a bare module name is allowed as the first argument to a "use".

Module name required with -%c option

(F) The `-M` or `-m` options say that Perl should load some module, but you omitted the name of the module. Consult [perlrun](#) for full details about `-M` and `-m`.

More than one argument to open

(F) The `open` function has been asked to open multiple files. This can happen if you are trying to open a pipe to a command that takes a list of arguments, but have forgotten to specify a piped open mode. See [open](#) for details.

msg%s not implemented

(F) You don't have System V message IPC on your system.

Multidimensional syntax %s not supported

(W syntax) Multidimensional arrays aren't written like `$foo[1,2,3]`. They're written like `$foo[1][2][3]`, as in C.

/ must be followed by a*, A* or Z*

(F) You had a pack template indicating a counted-length string. Currently the only things that can have their length counted are `a*`, `A*` or `Z*`. See [pack](#).

/ must be followed by a, A or Z

(F) You had an unpack template indicating a counted-length string, which must be followed by one of the letters `a`, `A` or `Z` to indicate what sort of string is to be unpacked. See [pack](#).

/ must follow a numeric type

(F) You had an unpack template that contained a `#`, but this did not follow some numeric unpack specification. See [pack](#).

"my sub" not yet implemented

(F) Lexically scoped subroutines are not yet implemented. Don't try that yet.

"my" variable %s can't be in a package

(F) Lexically scoped variables aren't in a package, so it doesn't make sense to try to declare one with a package qualifier on the front. Use `local()` if you want to localize a package variable.

Name "%s::%s" used only once: possible typo

(W once) Typographical errors often show up as unique variable names. If you had a good reason for having a unique name, then just mention it again somehow to suppress the message. The `our` declaration is provided for this purpose.

Negative length

(F) You tried to do a read/write/send/rcv operation with a buffer length that is less than 0. This is difficult to imagine.

Negative offset to vec in lvalue context

(F) When `vec` is called in an lvalue context, the second argument must be greater than or equal to zero.

Nested quantifiers in regex; marked by <— HERE in m/%s/

(F) You can't quantify a quantifier without intervening parentheses. So things like `**` or `+*` or `?*` are illegal. The <— HERE shows in the regular expression about where the problem was discovered.

Note that the minimal matching quantifiers, `*?`, `+?`, and `??` appear to be nested quantifiers, but aren't. See [perlre](#).

%s never introduced

(S internal) The symbol in question was declared but somehow went out of scope before it could possibly have been used.

No %s allowed while running setuid

(F) Certain operations are deemed to be too insecure for a setuid or setgid script to even be allowed to attempt. Generally speaking there will be another way to do what you want that is, if not secure, at least securable. See [perlsec](#).

No `-e` allowed in setuid scripts

(F) A setuid script can't be specified by the user.

No comma allowed after %s

(F) A list operator that has a filehandle or "indirect object" is not allowed to have a comma between that and the following arguments. Otherwise it'd be just another one of the arguments.

One possible cause for this is that you expected to have imported a constant to your name space with **use** or **import** while no such importing took place, it may for example be that your operating system does not support that particular constant. Hopefully you did use an explicit import list for the constants you expect to see, please see [use](#) and [import](#). While an explicit import list would probably have caught this error earlier it naturally does not remedy the fact that your operating system still does not support that constant. Maybe you have a typo in the constants of the symbol import list of **use** or **import** or in the constant name at the line where this error was triggered?

No command into which to pipe on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '|' at the end of the command line, so it doesn't know where you want to pipe the output from this command.

No DB::DB routine defined

(F) The currently executing code was compiled with the `-d` switch, but for some reason the `perl5db.pl` file (or some facsimile thereof) didn't define a routine to be called at the beginning of each statement. Which is odd, because the file should have been required automatically, and should have blown up the require if it didn't parse right.

No dbm on this machine

(P) This is counted as an internal error, because every machine should supply dbm nowadays, because Perl comes with SDBM. See [SDBM_File](#).

No DBsub routine

(F) The currently executing code was compiled with the `-d` switch, but for some reason the `perl5db.pl` file (or some facsimile thereof) didn't define a `DB::sub` routine to be called at the beginning of each ordinary subroutine call.

No error file after 2 or 2 on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '2' or a '2' on the command line, but can't find the name of the file to which to write data destined for stderr.

No input file after < on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a '<' on the command line, but can't find the name of the file from which to read data for stdin.

No #! line

(F) The `setuid` emulator requires that scripts have a well-formed `#!` line even on machines that don't support the `#!` construct.

"no" not allowed in expression

(F) The `"no"` keyword is recognized and executed at compile time, and returns no useful value. See [perlmod](#).

No output file after `on` command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a lone `'` at the end of the command line, so it doesn't know where you wanted to redirect `stdout`.

No output file after `or` on command line

(F) An error peculiar to VMS. Perl handles its own command line redirection, and found a `'` or a `'` on the command line, but can't find the name of the file to which to write data destined for `stdout`.

No package name allowed for variable `%s` in `"our"`

(F) Fully qualified variable names are not allowed in `"our"` declarations, because that doesn't make much sense under existing semantics. Such syntax is reserved for future extensions.

No Perl script found in input

(F) You called `perl -x`, but no line was found in the file beginning with `#!` and containing the word `"perl"`.

No `setregid` available

(F) Configure didn't find anything resembling the `setregid()` call for your system.

No `setreuid` available

(F) Configure didn't find anything resembling the `setreuid()` call for your system.

No space allowed after `-%c`

(F) The argument to the indicated command line switch must follow immediately after the switch, without intervening spaces.

No `%s` specified for `-%c`

(F) The indicated command line switch needs a mandatory argument, but you haven't specified one.

No such class `%s`

(F) You provided a class qualifier in a `"my"` or `"our"` declaration, but this class doesn't exist at this point in your program.

No such pipe open

(P) An error peculiar to VMS. The internal routine `my_pclose()` tried to close a pipe which hadn't been opened. This should have been caught earlier as an attempt to close an unopened filehandle.

No such pseudo-hash field `"%s"`

(F) You tried to access an array as a hash, but the field name used is not defined. The hash at index 0 should map all valid field names to array indices for that to work.

No such pseudo-hash field `"%s"` in variable `%s` of type `%s`

(F) You tried to access a field of a typed variable where the type does not know about the field name. The field names are looked up in the `%FIELDS` hash in the type package at compile time. The `%FIELDS` hash is usually set up with the `'fields'` pragma.

No such signal: `SIG%s`

(W signal) You specified a signal name as a subscript to `%SIG` that was not recognized. Say `kill -1` in your shell to see the valid signal names on your system.

Not a CODE reference

(F) Perl was trying to evaluate a reference to a code value (that is, a subroutine), but found a reference to something else instead. You can use the `ref()` function to find out what kind of ref it really was. See also [perlref](#).

Not a format reference

(F) I'm not sure how you managed to generate a reference to an anonymous format, but this indicates you did, and that it didn't exist.

Not a GLOB reference

(F) Perl was trying to evaluate a reference to a "typeglob" (that is, a symbol table entry that looks like `*foo`), but found a reference to something else instead. You can use the `ref()` function to find out what kind of ref it really was. See [perlref](#).

Not a HASH reference

(F) Perl was trying to evaluate a reference to a hash value, but found a reference to something else instead. You can use the `ref()` function to find out what kind of ref it really was. See [perlref](#).

Not an ARRAY reference

(F) Perl was trying to evaluate a reference to an array value, but found a reference to something else instead. You can use the `ref()` function to find out what kind of ref it really was. See [perlref](#).

Not a perl script

(F) The setuid emulator requires that scripts have a well-formed `#!` line even on machines that don't support the `#!` construct. The line must mention perl.

Not a SCALAR reference

(F) Perl was trying to evaluate a reference to a scalar value, but found a reference to something else instead. You can use the `ref()` function to find out what kind of ref it really was. See [perlref](#).

Not a subroutine reference

(F) Perl was trying to evaluate a reference to a code value (that is, a subroutine), but found a reference to something else instead. You can use the `ref()` function to find out what kind of ref it really was. See also [perlref](#).

Not a subroutine reference in overload table

(F) An attempt was made to specify an entry in an overloading table that doesn't somehow point to a valid subroutine. See [overload](#).

Not enough arguments for %s

(F) The function requires more arguments than you specified.

Not enough format arguments

(W syntax) A format specified more picture fields than the next line supplied. See [perlform](#).

%s: not found

(A) You've accidentally run your script through the Bourne shell instead of Perl. Check the `#!` line, or manually feed your script into Perl yourself.

%s not allowed in length fields

(F) The count in the (un)pack template may be replaced by `[TEMPLATE]` only if `TEMPLATE` always matches the same amount of packed bytes. Redesign the template.

no UTC offset information; assuming local time is UTC

(S) A warning peculiar to VMS. Perl was unable to find the local timezone offset, so it's assuming that local system time is equivalent to UTC. If it's not, define the logical name `SYS$TIMEZONE_DIFFERENTIAL` to translate to the number of seconds which need to be added to UTC to get local time.

Null filename used

(F) You can't require the null filename, especially because on many machines that means the current directory! See [require](#).

NULL OP IN RUN

(P debugging) Some internal routine called `run()` with a null opcode pointer.

Null picture in formline

(F) The first argument to `formline` must be a valid format picture specification. It was found to be empty, which probably means you supplied it an uninitialized value. See [perlfm](#).

Null realloc

(P) An attempt was made to `realloc` NULL.

NULL regexp argument

(P) The internal pattern matching routines blew it big time.

NULL regexp parameter

(P) The internal pattern matching routines are out of their gourd.

Number too long

(F) Perl limits the representation of decimal numbers in programs to about 250 characters. You've exceeded that length. Future versions of Perl are likely to eliminate this arbitrary limitation. In the meantime, try using scientific notation (e.g. "1e6" instead of "1_000_000").

Octal number in vector unsupported

(F) Numbers with a leading `o` are not currently allowed in vectors. The octal number interpretation of such numbers may be supported in a future version.

Octal number 037777777777 non-portable

(W portable) The octal number you specified is larger than $2^{32}-1$ (4294967295) and therefore non-portable between systems. See [perlport](#) for more on portability concerns.

See also [perlport](#) for writing portable code.

Odd number of arguments for overload::constant

(W overload) The call to `overload::constant` contained an odd number of arguments. The arguments should come in pairs.

Odd number of elements in anonymous hash

(W misc) You specified an odd number of elements to initialize a hash, which is odd, because hashes come in key/value pairs.

Odd number of elements in hash assignment

(W misc) You specified an odd number of elements to initialize a hash, which is odd, because hashes come in key/value pairs.

Offset outside string

(F) You tried to do a `read/write/send/rcv` operation with an offset pointing outside the buffer. This is difficult to imagine. The sole exception to this is that `sysread()` ing past the buffer will extend the buffer and zero pad the new area.

-%s on unopened filehandle %s

(W unopened) You tried to invoke a file test operator on a filehandle that isn't open. Check your control flow. See also [-X](#).

%s() on unopened %s

(W unopened) An I/O operation was attempted on a filehandle that was never initialized. You need to do an `open()`, a `sysopen()`, or a `socket()` call, or call a constructor from the `FileHandle`

- package.
- oops: oopsAV
(S internal) An internal warning that the grammar is screwed up.
- oops: oopsHV
(S internal) An internal warning that the grammar is screwed up.
- Operation '%s': no method found, %s
(F) An attempt was made to perform an overloaded operation for which no handler was defined. While some handlers can be autogenerated in terms of other handlers, there is no default handler for any operation, unless `fallback` overloading key is specified to be true. See [overload](#).
- Operator or semicolon missing before %s
(S ambiguous) You used a variable or subroutine call where the parser was expecting an operator. The parser has assumed you really meant to use an operator, but this is highly likely to be incorrect. For example, if you say `"*foo *foo"` it will be interpreted as if you said `"*foo * 'foo'"`.
- "our" variable %s redeclared
(W misc) You seem to have already declared the same global once before in the current lexical scope.
- Out of memory!
(X) The `malloc()` function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request. Perl has no option but to exit immediately.
- Out of memory during "large" request for %s
(F) The `malloc()` function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request. However, the request was judged large enough (compile-time default is 64K), so a possibility to shut down by trapping this error is granted.
- Out of memory during request for %s
(X|F) The `malloc()` function returned 0, indicating there was insufficient remaining memory (or virtual memory) to satisfy the request.
- The request was judged to be small, so the possibility to trap it depends on the way perl was compiled. By default it is not trappable. However, if compiled for this, Perl may use the contents of `$_M` as an emergency pool after `die()`ing with this message. In this case the error is trappable *once*, and the error message will include the line and file where the failed request happened.
- Out of memory during ridiculously large request
(F) You can't allocate more than $2^{31} + \text{"small amount"}$ bytes. This error is most likely to be caused by a typo in the Perl program. e.g., `$arr[time]` instead of `$arr[$time]`.
- Out of memory for yacc stack
(F) The yacc parser wanted to grow its stack so it could continue parsing, but `realloc()` wouldn't give it more memory, virtual or otherwise.
- @ outside of string
(F) You had a pack template that specified an absolute position outside the string being unpacked. See [pack](#).
- %s package attribute may clash with future reserved word: %s
(W reserved) A lowercase attribute name was used that had a package-specific handler. That name might have a meaning to Perl itself some day, even though it doesn't yet. Perhaps you should use a mixed-case attribute name, instead. See [attributes](#).
- page overflow
(W io) A single call to `write()` produced more lines than can fit on a page. See [perlform](#).

panic: %s

(P) An internal error.

panic: ck_grep

(P) Failed an internal consistency check trying to compile a grep.

panic: ck_split

(P) Failed an internal consistency check trying to compile a split.

panic: corrupt saved stack index

(P) The savestack was requested to restore more localized values than there are in the savestack.

panic: del_backref

(P) Failed an internal consistency check while trying to reset a weak reference.

panic: die %s

(P) We popped the context stack to an eval context, and then discovered it wasn't an eval context.

panic: pp_match%s

(P) The internal `pp_match()` routine was called with invalid operational data.

panic: do_subst

(P) The internal `pp_subst()` routine was called with invalid operational data.

panic: do_trans_%s

(P) The internal `do_trans` routines were called with invalid operational data.

panic: frexp

(P) The library function `frexp()` failed, making `printf("%f")` impossible.

panic: goto

(P) We popped the context stack to a context with the specified label, and then discovered it wasn't a context we know how to do a goto in.

panic: INTERPCASEMOD

(P) The lexer got into a bad state at a case modifier.

panic: INTERPCONCAT

(P) The lexer got into a bad state parsing a string with brackets.

panic: kid popen errno read

(F) forked child returned an incomprehensible message about its `errno`.

panic: last

(P) We popped the context stack to a block context, and then discovered it wasn't a block context.

panic: leave_scope clearsv

(P) A writable lexical variable became read-only somehow within the scope.

panic: leave_scope inconsistency

(P) The savestack probably got out of sync. At least, there was an invalid enum on the top of it.

panic: magic_killbackrefs

(P) Failed an internal consistency check while trying to reset all weak references to an object.

panic: malloc

(P) Something requested a negative number of bytes of `malloc`.

panic: mapstart

(P) The compiler is screwed up with respect to the `map ()` function.

panic: null array

(P) One of the internal array routines was passed a null AV pointer.

panic: pad_alloc

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_free curpad

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_free po

(P) An invalid scratch pad offset was detected internally.

panic: pad_reset curpad

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_sv po

(P) An invalid scratch pad offset was detected internally.

panic: pad_swipe curpad

(P) The compiler got confused about which scratch pad it was allocating and freeing temporaries and lexicals from.

panic: pad_swipe po

(P) An invalid scratch pad offset was detected internally.

panic: pp_iter

(P) The foreach iterator got called in a non-loop context frame.

panic: pp_split

(P) Something terrible went wrong in setting up for the split.

panic: realloc

(P) Something requested a negative number of bytes of realloc.

panic: restartop

(P) Some internal routine requested a goto (or something like it), and didn't supply the destination.

panic: return

(P) We popped the context stack to a subroutine or eval context, and then discovered it wasn't a subroutine or eval context.

panic: scan_num

(P) `scan_num ()` got called on something that wasn't a number.

panic: sv_insert

(P) The `sv_insert ()` routine was told to remove more string than there was string.

panic: top_env

(P) The compiler attempted to do a goto, or something weird like that.

panic: yylex

(P) The lexer got into a bad state while processing a case modifier.

panic: utf16_to_utf8: odd bytelen

(P) Something tried to call utf16_to_utf8 with an odd (as opposed to even) byte length.

Parentheses missing around "%s" list

(W parenthesis) You said something like

```
my $foo, $bar = @_;
```

when you meant

```
my ($foo, $bar) = @_;
```

Remember that "my", "our", and "local" bind tighter than comma.

Perl %s required—this is only version %s, stopped

(F) The module in question uses features of a version of Perl more recent than the currently running version. How long has it been since you upgraded, anyway? See [require](#).

PERL_SH_DIR too long

(F) An error peculiar to OS/2. PERL_SH_DIR is the directory to find the sh-shell in. See "PERL_SH_DIR" in [perlos2](#).

perl: warning: Setting locale failed.

(S) The whole warning message will look something like:

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
    LC_ALL = "En_US",
    LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

Exactly what were the failed locale settings varies. In the above the settings were that the LC_ALL was "En_US" and the LANG had no value. This error means that Perl detected that you and/or your operating system supplier and/or system administrator have set up the so-called locale system but Perl could not use those settings. This was not dead serious, fortunately: there is a "default locale" called "C" that Perl can and will use, the script will be run. Before you really fix the problem, however, you will get the same error message each time you run Perl. How to really fix the problem can be found in [perllocale](#) section **LOCALE PROBLEMS**.

perlio: argument list not closed for layer "%s"

(W layer) When pushing a layer with arguments onto the Perl I/O system you forgot the) that closes the argument list. (Layers take care of transforming data between external and internal representations.) Perl stopped parsing the layer list at this point and did not attempt to push this layer. If your program didn't explicitly request the failing operation, it may be the result of the value of the environment variable PERLIO.

perlio: invalid separator character %s in layer specification list %s

(W layer) When pushing layers onto the Perl I/O system, something other than a colon or whitespace was seen between the elements of a layer list. If the previous attribute had a parenthesised parameter list, perhaps that list was terminated too soon.

perlio: unknown layer "%s"

(W layer) An attempt was made to push an unknown layer onto the Perl I/O system. (Layers take care of transforming data between external and internal representations.) Note that some layers, such as mmap, are not supported in all environments. If your program didn't explicitly request the failing operation, it may be the result of the value of the environment variable PERLIO.

Permission denied

(F) The setuid emulator in `suidperl` decided you were up to no good.

pid %x not a child

(W exec) A warning peculiar to VMS. `Waitpid()` was asked to wait for a process which isn't a subprocess of the current process. While this is fine from VMS' perspective, it's probably not what you intended.

P must have an explicit size

(F) The unpack format P must have an explicit size, not `"*"`.

POSIX syntax [%s] belongs inside character classes in regex;

marked by `<—` HERE in `m/%s/`

(W regex) The character class constructs `[:]`, `[=]`, and `[.]` go *inside* character classes, the `[]` are part of the construct, for example: `/[012[:alpha:]]345]/`. Note that `[=]` and `[.]` are not currently implemented; they are simply placeholders for future extensions and will cause fatal errors. The `<—` HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

POSIX syntax [.] is reserved for future extensions in regex;

marked by `<—` HERE in `m/%s/`

(F regex) Within regular expression character classes (`[]`) the syntax beginning with `"["` and ending with `"]"` is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: `"\[."` and `"]"`. The `<—` HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

POSIX syntax [= =] is reserved for future extensions in regex;

marked by `<—` HERE in `m/%s/`

(F) Within regular expression character classes (`[]`) the syntax beginning with `"["` and ending with `"]"` is reserved for future extensions. If you need to represent those character sequences inside a regular expression character class, just quote the square brackets with the backslash: `"\[="` and `"]="`. The `<—` HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

POSIX class [%s:] unknown in regex;

marked by `<—` HERE in `m/%s/`

(F) The class in the character class `[:]` syntax is unknown. The `<—` HERE shows in the regular expression about where the problem was discovered. Note that the POSIX character classes do **not** have the `is` prefix the corresponding C interfaces have: in other words, it's `[[:print:]]`, not `isprint`. See [perlre](#).

POSIX getpgrp can't take an argument

(F) Your system has `POSIX getpgrp()`, which takes no argument, unlike the BSD version, which takes a pid.

Possible attempt to put comments in `qw()` list

(W qw) `qw()` lists contain items separated by whitespace; as with literal strings, comment characters are not ignored, but are instead treated as literal data. (You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
@list = qw(
    a # a comment
    b # another comment
);
```

when you should have written this:

```
@list = qw(
    a
    b
);
```

If you really want comments, build your list the old-fashioned way, with quotes and commas:

```
@list = (
    'a',      # a comment
    'b',      # another comment
);
```

Possible attempt to separate words with commas

(W qw) `qw()` lists contain items separated by whitespace; therefore commas aren't needed to separate the items. (You may have used different delimiters than the parentheses shown here; braces are also frequently used.)

You probably wrote something like this:

```
qw! a, b, c !;
```

which puts literal commas into some of the list items. Write it without commas if you don't want them to appear in your data:

```
qw! a b c !;
```

Possible memory corruption: %s overflowed 3rd argument

(F) An `ioctl()` or `fcntl()` returned more than Perl was bargaining for. Perl guesses a reasonable buffer size, but puts a sentinel byte at the end of the buffer just in case. This sentinel byte got clobbered, and Perl assumes that memory is now corrupted. See [ioctl](#).

Possible unintended interpolation of %s in string

(W ambiguous) You said something like '@foo' in a double-quoted string but there was no array `@foo` in scope at the time. If you wanted a literal `@foo`, then write it as `\@foo`; otherwise find out what happened to the array you apparently lost track of.

Possible Y2K bug: %s

(W y2k) You are concatenating the number 19 with another number, which could be a potential Year 2000 problem.

`pragma "attrs"` is deprecated, use `"sub NAME : ATTRS"` instead

(D deprecated) You have written something like this:

```
sub doit
{
    use attrs qw(locked);
}
```

You should use the new declaration syntax instead.

```
sub doit : locked
{
    ...
}
```

The `use attrs` pragma is now obsolete, and is only provided for backward-compatibility. See [Subroutine Attributes in perlsub](#).

Precedence problem: `open %s` should be `open(%s)`

(S precedence) The old irregular construct

```
open FOO || die;
```

is now misinterpreted as

```
open(FOO || die);
```

because of the strict regularization of Perl 5's grammar into unary and list operators. (The old open was a little of both.) You must put parentheses around the filehandle, or use the new "or" operator instead of "||".

Premature end of script headers

See Server error.

printf() on closed filehandle %s

(W closed) The filehandle you're writing to got itself closed sometime before now. Check your control flow.

print() on closed filehandle %s

(W closed) The filehandle you're printing on got itself closed sometime before now. Check your control flow.

Process terminated by SIG%s

(W) This is a standard message issued by OS/2 applications, while *nix applications die in silence. It is considered a feature of the OS/2 port. One can easily disable this by appropriate sighandlers, see [Signals in perlipc](#). See also "Process terminated by SIGTERM/SIGINT" in [perlos2](#).

Prototype mismatch: %s vs %s

(S prototype) The subroutine being declared or defined had previously been declared or defined with a different function prototype.

Prototype not terminated

(F) You've omitted the closing parenthesis in a function prototype definition.

Quantifier in {,} bigger than %d in regex;

marked by <— HERE in m/%s/

(F) There is currently a limit to the size of the min and max values of the {min,max} construct. The <— HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

Quantifier unexpected on zero-length expression;

marked by <— HERE in m/%s/

(W regexp) You applied a regular expression quantifier in a place where it makes no sense, such as on a zero-width assertion. Try putting the quantifier inside the assertion instead. For example, the way to match "abc" provided that it is followed by three repetitions of "xyz" is `/abc(?:xyz){3}/`, not `/abc(?:xyz){3}/`.

The <— HERE shows in the regular expression about where the problem was discovered.

Range iterator outside integer range

(F) One (or both) of the numeric arguments to the range operator ".." are outside the range which can be represented by integers internally. One possible workaround is to force Perl to use magical string increment by prepending "0" to your numbers.

readline() on closed filehandle %s

(W closed) The filehandle you're reading from got itself closed sometime before now. Check your control flow.

Reallocation too large: %lx

(F) You can't allocate more than 64K on an MS-DOS machine.

realloc() of freed memory ignored

(S malloc) An internal routine called `realloc()` on something that had already been freed.

Recompile perl with `-DDEBUGGING` to use `-D` switch

(F debugging) You can't use the `-D` option unless the code to produce the desired output is compiled into Perl, which entails some overhead, which is why it's currently left out of your copy.

Recursive inheritance detected in package '%s'

(F) More than 100 levels of inheritance were used. Probably indicates an unintended loop in your inheritance hierarchy.

Recursive inheritance detected while looking for method %s

(F) More than 100 levels of inheritance were encountered while invoking a method. Probably indicates an unintended loop in your inheritance hierarchy.

Reference found where even-sized list expected

(W misc) You gave a single reference where Perl was expecting a list with an even number of elements (for assignment to a hash). This usually means that you used the anon hash constructor when you meant to use parens. In any case, a hash requires key/value **pairs**.

```
%hash = { one => 1, two => 2, };      # WRONG
%hash = [ qw/ an anon array / ];    # WRONG
%hash = ( one => 1, two => 2, );      # right
%hash = qw( one 1 two 2 );           # also fine
```

Reference is already weak

(W misc) You have attempted to weaken a reference that is already weak. Doing so has no effect.

Reference miscount in `sv_replace()`

(W internal) The internal `sv_replace()` function was handed a new SV with a reference count of other than 1.

Reference to nonexistent group in regex;

marked by <— HERE in m/%s/

(F) You used something like `\7` in your regular expression, but there are not at least seven sets of capturing parentheses in the expression. If you wanted to have the character with value 7 inserted into the regular expression, prepend a zero to make the number at least two digits: `\07`

The <— HERE shows in the regular expression about where the problem was discovered.

regex memory corruption

(P) The regular expression engine got confused by what the regular expression compiler gave it.

Regex out of space

(P) A "can't happen" error, because `safemalloc()` should have caught it earlier.

Repeat count in pack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See [pack](#).

Repeat count in unpack overflows

(F) You can't specify a repeat count so large that it overflows your signed integers. See [unpack](#).

Reversed `%s=` operator

(W syntax) You wrote your assignment operator backwards. The `=` must always comes last, to avoid ambiguity with subsequent unary operators.

Runaway format

(F) Your format contained the `~~` repeat-until-blank sequence, but it produced 200 lines at once, and the 200th line looked exactly like the 199th line. Apparently you didn't arrange for the arguments to exhaust themselves, either by using `^` instead of `@` (for scalar variables), or by shifting or popping (for array variables). See [perlform](#).

Scalar value `@%s[%s]` better written as `$$s[$s]`

(W syntax) You've used an array slice (indicated by `@`) to select a single element of an array. Generally it's better to ask for a scalar value (indicated by `$`). The difference is that `$foo[&bar]` always behaves like a scalar, both when assigning to it and when evaluating its argument, while `@foo[&bar]` behaves like a list when you assign to it, and provides a list context to its subscript, which can do weird things if you're expecting only one subscript.

On the other hand, if you were actually hoping to treat the array element as a list, you need to look into how references work, because Perl will not magically convert between scalars and lists for you. See [perlref](#).

Scalar value `@%s{%s}` better written as `$$s{%s}`

(W syntax) You've used a hash slice (indicated by `@`) to select a single element of a hash. Generally it's better to ask for a scalar value (indicated by `$`). The difference is that `$foo{&bar}` always behaves like a scalar, both when assigning to it and when evaluating its argument, while `@foo{&bar}` behaves like a list when you assign to it, and provides a list context to its subscript, which can do weird things if you're expecting only one subscript.

On the other hand, if you were actually hoping to treat the hash element as a list, you need to look into how references work, because Perl will not magically convert between scalars and lists for you. See [perlref](#).

Scalars leaked: `%d`

(P) Something went wrong in Perl's internal bookkeeping of scalars: not all scalar variables were deallocated by the time Perl exited. What this usually indicates is a memory leak, which is of course bad, especially if the Perl program is intended to be long-running.

Script is not `setuid/setgid` in `suidperl`

(F) Oddly, the `suidperl` program was invoked on a script without a `setuid` or `setgid` bit set. This doesn't make much sense.

Search pattern not terminated

(F) The lexer couldn't find the final delimiter of a `//` or `m{}` construct. Remember that bracketing delimiters count nesting level. Missing the leading `$` from a variable `$m` may cause this error.

`%sseek()` on unopened filehandle

(W unopened) You tried to use the `seek()` or `sysseek()` function on a filehandle that was either never opened or has since been closed.

`select` not implemented

(F) This machine doesn't implement the `select()` system call.

Self-ties of arrays and hashes are not supported

(F) Self-ties of arrays and hashes are not supported in the current implementation.

Semicolon seems to be missing

(W semicolon) A nearby syntax error was probably caused by a missing semicolon, or possibly some other missing operator, such as a comma.

semi-panic: attempt to dup freed string

(S internal) The internal `newSVsv()` routine was called to duplicate a scalar that had previously been marked as free.

sem%s not implemented

(F) You don't have System V semaphore IPC on your system.

send () on closed socket %s

(W closed) The socket you're sending to got itself closed sometime before now. Check your control flow.

Sequence (? incomplete in regex; marked by <— HERE in m/%s/

(F) A regular expression ended with an incomplete extension (? The <— HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

Sequence (?{...}) not terminated or not {}-balanced in regex;

marked by <— HERE in m/%s/

(F) If the contents of a (?{...}) clause contains braces, they must balance for Perl to properly detect the end of the clause. The <— HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

Sequence (?%s...) not implemented in regex;

marked by <— HERE in m/%s/

(F) A proposed regular expression extension has the character reserved but has not yet been written. The <— HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

Sequence (?%s...) not recognized in regex;

marked by <— HERE in m/%s/

(F) You used a regular expression extension that doesn't make sense. The <— HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

Sequence (?#... not terminated in regex;

marked by <— HERE in m/%s/

(F) A regular expression comment must be terminated by a closing parenthesis. Embedded parentheses aren't allowed. The <— HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

500 Server error

See Server error.

Server error

This is the error message generally seen in a browser window when trying to run a CGI program (including SSI) over the web. The actual error text varies widely from server to server. The most frequently-seen variants are "500 Server error", "Method (something) not permitted", "Document contains no data", "Premature end of script headers", and "Did not produce a valid header".

This is a CGI error, not a Perl error.

You need to make sure your script is executable, is accessible by the user CGI is running the script under (which is probably not the user account you tested it under), does not rely on any environment variables (like PATH) from the user it isn't running under, and isn't in a location where the CGI server can't find it, basically, more or less. Please see the following for more information:

```
http://www.perl.org/CGI_MetaFAQ.html
http://www.htmlhelp.org/faq/cgifaq.html
http://www.w3.org/Security/Faq/
```

You should also look at [perlfaq9](#).

setegid() not implemented

(F) You tried to assign to `$` , and your operating system doesn't support the `setegid()` system call (or equivalent), or at least Configure didn't think so.

seteuid() not implemented

(F) You tried to assign to `< $` , and your operating system doesn't support the `seteuid()` system call (or equivalent), or at least Configure didn't think so.

setpgrp can't take arguments

(F) Your system has the `setpgrp()` from BSD 4.2, which takes no arguments, unlike POSIX `setpgid()` , which takes a process ID and process group ID.

setrgid() not implemented

(F) You tried to assign to `$ (` , and your operating system doesn't support the `setrgid()` system call (or equivalent), or at least Configure didn't think so.

setruid() not implemented

(F) You tried to assign to `$<` , and your operating system doesn't support the `setruid()` system call (or equivalent), or at least Configure didn't think so.

setsockopt() on closed socket %s

(W closed) You tried to set a socket option on a closed socket. Did you forget to check the return value of your `socket()` call? See [setsockopt](#).

Setuid/gid script is writable by world

(F) The `setuid` emulator won't run a script that is writable by the world, because the world might have written on it already.

shm%s not implemented

(F) You don't have System V shared memory IPC on your system.

< should be quotes

(F) You wrote `< require <file` when you should have written `require 'file'`.

/%/s/ should probably be written as "%s"

(W syntax) You have used a pattern where Perl expected to find a string, as in the first argument to `join`. Perl will treat the true or false result of matching the pattern against `$_` as the string, which is probably not what you had in mind.

shutdown() on closed socket %s

(W closed) You tried to do a shutdown on a closed socket. Seems a bit superfluous.

SIG%s handler "%s" not defined

(W signal) The signal handler named in `%SIG` doesn't, in fact, exist. Perhaps you put it into the wrong package?

sort is now a reserved word

(F) An ancient error message that almost nobody ever runs into anymore. But before `sort` was a keyword, people sometimes used it as a filehandle.

Sort subroutine didn't return a numeric value

(F) A sort comparison routine must return a number. You probably blew it by not using `<` `<=` or `cmp`, or by not using them correctly. See [sort](#).

Sort subroutine didn't return single value

(F) A sort comparison subroutine may not return a list value with more or less than one element. See [sort](#).

splice() offset past end of array

(W misc) You attempted to specify an offset that was past the end of the array passed to `splice()`. Splicing will instead commence at the end of the array, rather than past it. If this isn't what you want, try explicitly pre-extending the array by assigning `$#array = $offset`. See [splice](#).

Split loop

(P) The split was looping infinitely. (Obviously, a split shouldn't iterate more times than there are characters of input, which is what happened.) See [split](#).

Statement unlikely to be reached

(W exec) You did an `exec()` with some statement after it other than a `die()`. This is almost always an error, because `exec()` never returns unless there was a failure. You probably wanted to use `system()` instead, which does return. To suppress this warning, put the `exec()` in a block by itself.

stat() on unopened filehandle %s

(W unopened) You tried to use the `stat()` function on a filehandle that was either never opened or has since been closed.

Stub found while resolving method '%s' overloading %s

(P) Overloading resolution over `@ISA` tree may be broken by importation stubs. Stubs should never be implicitly created, but explicit calls to `can` may break this.

Subroutine %s redefined

(W redefine) You redefined a subroutine. To suppress this warning, say

```
{
    no warnings 'redefine';
    eval "sub name { ... }";
}
```

Substitution loop

(P) The substitution was looping infinitely. (Obviously, a substitution shouldn't iterate more times than there are characters of input, which is what happened.) See the discussion of substitution in [Quote and Quote-like Operators in perlop](#).

Substitution pattern not terminated

(F) The lexer couldn't find the interior delimiter of an `s///` or `s{ }{ }` construct. Remember that bracketing delimiters count nesting level. Missing the leading `$` from variable `$s` may cause this error.

Substitution replacement not terminated

(F) The lexer couldn't find the final delimiter of an `s///` or `s{ }{ }` construct. Remember that bracketing delimiters count nesting level. Missing the leading `$` from variable `$s` may cause this error.

substr outside of string

(W substr),(F) You tried to reference a `substr()` that pointed outside of a string. That is, the absolute value of the offset was larger than the length of the string. See [substr](#). This warning is fatal if `substr` is used in an lvalue context (as the left hand side of an assignment or as a subroutine argument for example).

suidperl is no longer needed since %s

(F) Your Perl was compiled with `-DSETUID_SCRIPTS_ARE_SECURE_NOW`, but a version of the setuid emulator somehow got run anyway.

Switch (?(condition)... contains too many branches in regex;

marked by <— HERE in m/%s/

(F) A `(?(condition)if-clause|else-clause)` construct can have at most two branches (the `if-clause` and the `else-clause`). If you want one or both to contain alternation, such as using `this|that|other`, enclose it in clustering parentheses:

```
(?(condition) (?:this|that|other) |else-clause)
```

The `<—` HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

Switch condition not recognized in regex;

marked by `<—` HERE in `m/%s/`

(F) If the argument to the `(?(...))if-clause|else-clause)` construct is a number, it can be only a number. The `<—` HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

switching effective %s is not implemented

(F) While under the `use filetest` pragma, we cannot switch the real and effective uids or gids.

syntax error

(F) Probably means you had a syntax error. Common reasons include:

- A keyword is misspelled.
- A semicolon is missing.
- A comma is missing.
- An opening or closing parenthesis is missing.
- An opening or closing brace is missing.
- A closing quote is missing.

Often there will be another error message associated with the syntax error giving more information. (Sometimes it helps to turn on `-w`.) The error message itself often tells you where it was in the line when it decided to give up. Sometimes the actual error is several tokens before this, because Perl is good at understanding random input. Occasionally the line number may be misleading, and once in a blue moon the only way to figure out what's triggering the error is to call `perl -c` repeatedly, chopping away half the program each time to see if the error went away. Sort of the cybernetic version of 20 questions.

syntax error at line %d: '%s' unexpected

(A) You've accidentally run your script through the Bourne shell instead of Perl. Check the `#!` line, or manually feed your script into Perl yourself.

syntax error in file %s at line %d, next 2 tokens "%s"

(F) This error is likely to occur if you run a perl5 script through a perl4 interpreter, especially if the next 2 tokens are `"use strict"` or `"my $var"` or `"our $var"`.

%s syntax OK

(F) The final summary message when a `perl -c` succeeds.

System V %s is not implemented on this machine

(F) You tried to do something with a function beginning with `"sem"`, `"shm"`, or `"msg"` but that System V IPC is not implemented in your machine. In some machines the functionality can exist but be unconfigured. Consult your system support.

`syswrite()` on closed filehandle %s

(W closed) The filehandle you're writing to got itself closed sometime before now. Check your control flow.

Target of goto is too deeply nested

(F) You tried to use `goto` to reach a label that was too deeply nested for Perl to reach. Perl is doing you a favor by refusing.

tell() on unopened filehandle

(W unopened) You tried to use the `tell()` function on a filehandle that was either never opened or has since been closed.

That use of `$[` is unsupported

(F) Assignment to `$[` is now strictly circumscribed, and interpreted as a compiler directive. You may say only one of

```
$[ = 0;
$[ = 1;
...
local $[ = 0;
local $[ = 1;
...
```

This is to prevent the problem of one module changing the array base out from under another module inadvertently. See [\\$\[](#).

The `crypt()` function is unimplemented due to excessive paranoia

(F) Configure couldn't find the `crypt()` function on your machine, probably because your vendor didn't supply it, probably because they think the U.S. Government thinks it's a secret, or at least that they will continue to pretend that it is. And if you quote me on that, I will deny it.

The `%s` function is unimplemented

The function indicated isn't implemented on this architecture, according to the probings of Configure.

The stat preceding `%s` wasn't an `lstat`

(F) It makes no sense to test the current stat buffer for symbolic linkhood if the last stat that wrote to the stat buffer already went past the symlink to get to the real file. Use an actual filename instead.

This Perl can't reset CRTL environ elements (`%s`)**This Perl can't set CRTL environ elements (`%s=%s`)**

(W internal) Warnings peculiar to VMS. You tried to change or delete an element of the CRTL's internal environ array, but your copy of Perl wasn't built with a CRTL that contained the `setenv()` function. You'll need to rebuild Perl with a CRTL that does, or redefine ***PERL_ENV_TABLES*** (see [perlvms](#)) so that the environ array isn't the target of the change to `%ENV` which produced the warning.

thread failed to start: `%s`

(F) The entry point function of `threads-create()` failed for some reason.

times not implemented

(F) Your version of the C library apparently doesn't do `times()`. I suspect you're not running on Unix.

Too few args to `syscall`

(F) There has to be at least one argument to `syscall()` to specify the system call to call, silly dilly.

Too late for "`-T`" option

(X) The `#!` line (or local equivalent) in a Perl script contains the `-T` option, but Perl was not invoked with `-T` in its command line. This is an error because, by the time Perl discovers a `-T` in a script, it's too late to properly taint everything from the environment. So Perl gives up.

If the Perl script is being executed as a command using the `#!` mechanism (or its local equivalent), this error can usually be fixed by editing the `#!` line so that the `-T` option is a part of Perl's first argument: e.g. change `perl -n -T to perl -T -n`.

If the Perl script is being executed as `perl scriptname`, then the `-T` option must appear on the command line: `perl -T scriptname`.

Too late for "-%s" option

(X) The `#!` line (or local equivalent) in a Perl script contains the `-M` or `-m` option. This is an error because `-M` and `-m` options are not intended for use inside scripts. Use the `use` pragma instead.

Too late to run %s block

(W void) A `CHECK` or `INIT` block is being defined during run time proper, when the opportunity to run them has already passed. Perhaps you are loading a file with `require` or `do` when you should be using `use` instead. Or perhaps you should put the `require` or `do` inside a `BEGIN` block.

Too many args to syscall

(F) Perl supports a maximum of only 14 args to `syscall()`.

Too many arguments for %s

(F) The function requires fewer arguments than you specified.

Too many)'s**Too many ('s**

(A) You've accidentally run your script through `csh` instead of Perl. Check the `#!` line, or manually feed your script into Perl yourself.

Trailing \ in regex m/%s/

(F) The regular expression ends with an unbackslashed backslash. Backslash it. See [perlre](#).

Transliteration pattern not terminated

(F) The lexer couldn't find the interior delimiter of a `tr///` or `tr[][]` or `y///` or `y[][]` construct. Missing the leading `$` from variables `$tr` or `$y` may cause this error.

Transliteration replacement not terminated

(F) The lexer couldn't find the final delimiter of a `tr///` or `tr[][]` construct.

truncate not implemented

(F) Your machine doesn't implement a file truncation mechanism that Configure knows about.

Type of arg %d to %s must be %s (not %s)

(F) This function requires the argument in that position to be of a certain type. Arrays must be `@NAME` or `@{EXPR}`. Hashes must be `%NAME` or `%{EXPR}`. No implicit dereferencing is allowed—use the `{EXPR}` forms as an explicit dereference. See [perlref](#).

umask not implemented

(F) Your machine doesn't implement the `umask` function and you tried to use it to restrict permissions for yourself (`EXPR & 0700`).

Unable to create sub named "%s"

(F) You attempted to create or access a subroutine with an illegal name.

Unbalanced context: %d more PUSHes than POPs

(W internal) The exit code detected an internal inconsistency in how many execution contexts were entered and left.

Unbalanced saves: %d more saves than restores

(W internal) The exit code detected an internal inconsistency in how many values were temporarily localized.

Unbalanced scopes: %d more ENTERs than LEAVEs

(W internal) The exit code detected an internal inconsistency in how many blocks were entered and left.

Unbalanced tmps: %d more allocs than frees

(W internal) The exit code detected an internal inconsistency in how many mortal scalars were allocated and freed.

Undefined format "%s" called

(F) The format indicated doesn't seem to exist. Perhaps it's really in another package? See [perlform](#).

Undefined sort subroutine "%s" called

(F) The sort comparison routine specified doesn't seem to exist. Perhaps it's in a different package? See [sort](#).

Undefined subroutine &%s called

(F) The subroutine indicated hasn't been defined, or if it was, it has since been undefined.

Undefined subroutine called

(F) The anonymous subroutine you're trying to call hasn't been defined, or if it was, it has since been undefined.

Undefined subroutine in sort

(F) The sort comparison routine specified is declared but doesn't seem to have been defined yet. See [sort](#).

Undefined top format "%s" called

(F) The format indicated doesn't seem to exist. Perhaps it's really in another package? See [perlform](#).

Undefined value assigned to typeglob

(W misc) An undefined value was assigned to a typeglob, a la `*foo = undef`. This does nothing. It's possible that you really mean `undef *foo`.

%s: Undefined variable

(A) You've accidentally run your script through **csh** instead of Perl. Check the `#!` line, or manually feed your script into Perl yourself.

unexec of %s into %s failed!

(F) The `unexec()` routine failed for some reason. See your local FSF representative, who probably put it there in the first place.

Unicode character %s is illegal

(W utf8) Certain Unicode characters have been designated off-limits by the Unicode standard and should not be generated. If you really know what you are doing you can turn off this warning by `no warnings 'utf8';`.

Unknown BYTEORDER

(F) There are no byte-swapping functions for a machine with this byte order.

Unknown "re" subpragma '%s' (known ones are: %s)

You tried to use an unknown subpragma of the "re" pragma.

Unknown switch condition (?(%2s in regex;

marked by <— HERE in m/%s/

(F) The condition part of a `(?(condition)if-clause|else-clause)` construct is not known. The condition may be lookahead or lookbehind (the condition is true if the lookahead or lookbehind is true), a `(?{...})` construct (the condition is true if the code evaluates to a true value), or a number (the condition is true if the set of capturing parentheses named by the number matched).

The <— HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

Unrecognized switch: -%s (-h will show valid options)

(F) You specified an illegal option to Perl. Don't do that. (If you think you didn't do that, check the #! line to see if it's supplying the bad switch on your behalf.)

Unsuccessful %s on filename containing newline

(W newline) A file operation was attempted on a filename, and that operation failed, PROBABLY because the filename contained a newline, PROBABLY because you forgot to `chomp()` it off. See [chomp](#).

Unsupported directory function "%s" called

(F) Your machine doesn't support `opendir()` and `readdir()`.

Unsupported function %s

(F) This machine doesn't implement the indicated function, apparently. At least, Configure doesn't think so.

Unsupported function fork

(F) Your version of executable does not support forking.

Note that under some systems, like OS/2, there may be different flavors of Perl executables, some of which may support fork, some not. Try changing the name you call Perl by to `perl_`, `perl__`, and so on.

Unsupported script encoding

(F) Your program file begins with a Unicode Byte Order Mark (BOM) which declares it to be in a Unicode encoding that Perl cannot yet read.

Unsupported socket function "%s" called

(F) Your machine doesn't support the Berkeley socket mechanism, or at least that's what Configure thought.

Unterminated attribute list

(F) The lexer found something other than a simple identifier at the start of an attribute, and it wasn't a semicolon or the start of a block. Perhaps you terminated the parameter list of the previous attribute too soon. See [attributes](#).

Unterminated attribute parameter in attribute list

(F) The lexer saw an opening (left) parenthesis character while parsing an attribute list, but the matching closing (right) parenthesis character was not found. You may need to add (or remove) a backslash character to get your parentheses to balance. See [attributes](#).

Unterminated compressed integer

(F) An argument to `unpack("w",...)` was incompatible with the BER compressed integer format and could not be converted to an integer. See [pack](#).

Unterminated < operator

(F) The lexer saw a left angle bracket in a place where it was expecting a term, so it's looking for the corresponding right angle bracket, and not finding it. Chances are you left some needed parentheses out earlier in the line, and you really meant a "less than".

untie attempted while %d inner references still exist

(W untie) A copy of the object returned from `tie` (or `tied`) was still valid when `untie` was called.

Useless (?%s) – use /%s modifier in regex;

marked by <— HERE in `m/%s/`

(W regexp) You have used an internal modifier such as `(?o)` that has no meaning unless applied to the entire regexp:

```
if ($string =~ /(?)$pattern/) { ... }
```

must be written as

```
if ($string =~ /$pattern/) { ... }
```

The `<—` HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

Useless `(?-%s)` – don't use `/%s` modifier in regex;

marked by `<—` HERE in `m/%s/`

(W regexp) You have used an internal modifier such as `(?-o)` that has no meaning unless removed from the entire regexp:

```
if ($string =~ /(?-o)$pattern/o) { ... }
```

must be written as

```
if ($string =~ /$pattern/) { ... }
```

The `<—` HERE shows in the regular expression about where the problem was discovered. See [perlre](#).

Useless use of `%s` in void context

(W void) You did something without a side effect in a context that does nothing with the return value, such as a statement that doesn't return a value from a block, or the left side of a scalar comma operator.

Very often this points not to stupidity on your part, but a failure of Perl to parse your program the way you thought it would. For example, you'd get this if you mixed up your C precedence with Python precedence and said

```
$one, $two = 1, 2;
```

when you meant to say

```
($one, $two) = (1, 2);
```

Another common error is to use ordinary parentheses to construct a list reference when you should be using square or curly brackets, for example, if you say

```
$array = (1,2);
```

when you should have said

```
$array = [1,2];
```

The square brackets explicitly turn a list value into a scalar value, while parentheses do not. So when a parenthesized list is evaluated in a scalar context, the comma is treated like C's comma operator, which throws away the left argument, which is not what you want. See [perlref](#) for more on this.

This warning will not be issued for numerical constants equal to 0 or 1 since they are often used in statements like

```
1 while sub_with_side_effects() ;
```

String constants that would normally evaluate to 0 or 1 are warned about.

Useless use of "re" pragma

(W) You did use `re`; without any arguments. That isn't very useful.

Useless use of `sort` in scalar context

(W void) You used `sort` in scalar context, as in :

```
my $x = sort @y;
```

This is not very useful, and perl currently optimizes this away.

Useless use of %s with no values

(W syntax) You used the `push()` or `unshift()` function with no arguments apart from the array, like `push(@x)` or `unshift(@foo)`. That won't usually have any effect on the array, so is completely useless. It's possible in principle that `push(@tied_array)` could have some effect if the array is tied to a class which implements a `PUSH` method. If so, you can write it as `push(@tied_array, ())` to avoid this warning.

"use" not allowed in expression

(F) The "use" keyword is recognized and executed at compile time, and returns no useful value. See [perlmod](#).

Use of bare << to mean <<"" is deprecated

(D deprecated) You are now encouraged to use the explicitly quoted form if you wish to use an empty line as the terminator of the here-document.

Use of /c modifier is meaningless in s///

(W regexp) You used the `/c` modifier in a substitution. The `/c` modifier is not presently meaningful in substitutions.

Use of /c modifier is meaningless without /g

(W regexp) You used the `/c` modifier with a regex operand, but didn't use the `/g` modifier. Currently, `/c` is meaningful only when `/g` is used. (This may change in the future.)

Use of /g modifier is meaningless in split

(W regexp) You used the `/g` modifier on the pattern for a `split` operator. Since `split` always tries to match the pattern repeatedly, the `/g` has no effect.

Use of *glob{FILEHANDLE} is deprecated

(D deprecated) You are now encouraged to use the shorter `*glob{IO}` form to access the filehandle slot within a `typeglob`.

Use of chdir("") or chdir(undef) as chdir() deprecated

(D deprecated) `chdir()` with no arguments is documented to change to `$ENV{HOME}` or `$ENV{LOGDIR}`. `chdir(undef)` and `chdir("")` share this behavior, but that has been deprecated. In future versions they will simply fail.

Be careful to check that what you pass to `chdir()` is defined and not blank, else you might find yourself in your home directory.

Use of implicit split to @_ is deprecated

(D deprecated) It makes a lot of work for the compiler when you clobber a subroutine's argument list, so it's better if you assign the results of a `split()` explicitly to an array (or list).

Use of inherited AUTOLOAD for non-method %s() is deprecated

(D deprecated) As an (ahem) accidental feature, `AUTOLOAD` subroutines are looked up as methods (using the `@ISA` hierarchy) even when the subroutines to be autoloaded were called as plain functions (e.g. `Foo::bar()`), not as methods (e.g. `< Foo-bar()` or `< $obj-bar()`).

This bug will be rectified in future by using method lookup only for methods' `AUTOLOAD`s. However, there is a significant base of existing code that may be using the old behavior. So, as an interim step, Perl currently issues an optional warning when non-methods use inherited `AUTOLOAD`s.

The simple rule is: Inheritance will not work when autoloading non-methods. The simple fix for old code is: In any module that used to depend on inheriting `AUTOLOAD` for non-methods from a base class named `BaseClass`, execute `*AUTOLOAD = \&BaseClass::AUTOLOAD` during startup.

In code that currently says `use AutoLoader; @ISA = qw(AutoLoader);` you should remove `AutoLoader` from `@ISA` and change `use AutoLoader;` to `AutoLoader 'AUTOLOAD';`.

Use of `–l` on filehandle `%s`

(W io) A filehandle represents an opened file, and when you opened the file it already went past any symlink you are presumably trying to look for. The operation returned `undef`. Use a filename instead.

Use of `"package"` with no arguments is deprecated

(D deprecated) You used the `package` keyword without specifying a package name. So no namespace is current at all. Using this can cause many otherwise reasonable constructs to fail in baffling ways. use `strict`; instead.

Use of `%s` in `printf` format not supported

(F) You attempted to use a feature of `printf` that is accessible from only C. This usually means there's a better way to do it in Perl.

Use of `$*` is deprecated

(D deprecated) This variable magically turned on multi-line pattern matching, both for you and for any luckless subroutine that you happen to call. You should use the new `//m` and `//s` modifiers now to do that without the dangerous action-at-a-distance effects of `$*`.

Use of `%s` is deprecated

(D deprecated) The construct indicated is no longer recommended for use, generally because there's a better way to do it, and also because the old way has bad side effects.

Use of `$#` is deprecated

(D deprecated) This was an ill-advised attempt to emulate a poorly defined **awk** feature. Use an explicit `printf()` or `sprintf()` instead.

Use of reference `"%s"` as array index

(W misc) You tried to use a reference as an array index; this probably isn't what you mean, because references in numerical context tend to be huge numbers, and so usually indicates programmer error.

If you really do mean it, explicitly numify your reference, like so: `$array[0+$ref]`. This warning is not given for overloaded objects, either, because you can overload the numification and stringification operators and then you assumedly know what you are doing.

Use of reserved word `"%s"` is deprecated

(D deprecated) The indicated bareword is a reserved word. Future versions of perl may use it as a keyword, so you're better off either explicitly quoting the word in a manner appropriate for its context of use, or using a different name altogether. The warning can be suppressed for subroutine names by either adding a `&` prefix, or using a package qualifier, e.g. `&our()`, or `Foo::our()`.

Use of tainted arguments in `%s` is deprecated

(W taint, deprecated) You have supplied `system()` or `exec()` with multiple arguments and at least one of them is tainted. This used to be allowed but will become a fatal error in a future version of perl. Untaint your arguments. See [perlsec](#).

Use of uninitialized value `%s`

(W uninitialized) An undefined value was used as if it were already defined. It was interpreted as a "" or a 0, but maybe it was a mistake. To suppress this warning assign a defined value to your variables.

To help you figure out what was undefined, perl tells you what operation you used the undefined value in. Note, however, that perl optimizes your program and the operation displayed in the warning may not necessarily appear literally in your program. For example, `"that $foo"` is usually optimized into `"that " . $foo`, and the warning will refer to the concatenation `(.)` operator, even though there is no `.` in your program.

Using a hash as a reference is deprecated

(D deprecated) You tried to use a hash as a reference, as in `< %foo-{"bar"}` or `< %$ref-{"hello"}`. Versions of perl <= 5.6.1 used to allow this syntax, but shouldn't have. It is now deprecated, and will be removed in a future version.

Using an array as a reference is deprecated

(D deprecated) You tried to use an array as a reference, as in `< @foo-[23]` or `< @$ref-[99]`. Versions of perl <= 5.6.1 used to allow this syntax, but shouldn't have. It is now deprecated, and will be removed in a future version.

UTF-16 surrogate %s

(W utf8) You tried to generate half of an UTF-16 surrogate by requesting a Unicode character between the code points 0xD800 and 0xDFFF (inclusive). That range is reserved exclusively for the use of UTF-16 encoding (by having two 16-bit UCS-2 characters); but Perl encodes its characters in UTF-8, so what you got is a very illegal character. If you really know what you are doing you can turn off this warning by `no warnings 'utf8';`.

Value of %s can be "0"; test with defined()

(W misc) In a conditional expression, you used `<HANDLE`, `<*(glob)`, `each()`, or `readdir()` as a boolean value. Each of these constructs can return a value of "0"; that would make the conditional expression false, which is probably not what you intended. When using these constructs in conditional expressions, test their values with the `defined` operator.

Value of CLI symbol "%s" too long

(W misc) A warning peculiar to VMS. Perl tried to read the value of an %ENV element from a CLI symbol table, and found a resultant string longer than 1024 characters. The return value has been truncated to 1024 characters.

Variable "%s" is not imported%s

(F) While "use strict" in effect, you referred to a global variable that you apparently thought was imported from another module, because something else of the same name (usually a subroutine) is exported by that module. It usually means you put the wrong funny character on the front of your variable.

"%s" variable %s masks earlier declaration in same %s

(W misc) A "my" or "our" variable has been redeclared in the current scope or statement, effectively eliminating all access to the previous instance. This is almost always a typographical error. Note that the earlier variable will still exist until the end of the scope or until all closure referents to it are destroyed.

Variable "%s" may be unavailable

(W closure) An inner (nested) *anonymous* subroutine is inside a *named* subroutine, and outside that is another subroutine; and the anonymous (innermost) subroutine is referencing a lexical variable defined in the outermost subroutine. For example:

```
sub outermost { my $a; sub middle { sub { $a } } }
```

If the anonymous subroutine is called or referenced (directly or indirectly) from the outermost subroutine, it will share the variable as you would expect. But if the anonymous subroutine is called or referenced when the outermost subroutine is not active, it will see the value of the shared variable as it was before and during the **first** call to the outermost subroutine, which is probably not what you want.

In these circumstances, it is usually best to make the middle subroutine anonymous, using the `sub { }` syntax. Perl has specific support for shared variables in nested anonymous subroutines; a named subroutine in between interferes with this feature.

Variable syntax

(A) You've accidentally run your script through **csh** instead of Perl. Check the **#!** line, or manually feed your script into Perl yourself.

Variable "%s" will not stay shared

(W closure) An inner (nested) *named* subroutine is referencing a lexical variable defined in an outer subroutine.

When the inner subroutine is called, it will probably see the value of the outer subroutine's variable as it was before and during the **first** call to the outer subroutine; in this case, after the first call to the outer subroutine is complete, the inner and outer subroutines will no longer share a common value for the variable. In other words, the variable will no longer be shared.

Furthermore, if the outer subroutine is anonymous and references a lexical variable outside itself, then the outer and inner subroutines will *never* share the given variable.

This problem can usually be solved by making the inner subroutine anonymous, using the `sub { }` syntax. When inner anonymous subs that reference variables in outer subroutines are called or referenced, they are automatically rebound to the current values of such variables.

Variable length lookbehind not implemented in regex;

marked by `<— HERE` in `m/%s/`

(F) Lookbehind is allowed only for subexpressions whose length is fixed and known at compile time. The `<— HERE` shows in the regular expression about where the problem was discovered. See [perlre](#).

Version number must be a constant number

(P) The attempt to translate a `use Module n.n LIST` statement into its equivalent `BEGIN` block found an internal inconsistency with the version number.

v-string in use/require is non-portable

(W portable) The use of v-strings is non-portable to older, pre-5.6, Perls. If you want your scripts to be backward portable, use the floating point version number: for example, instead of `use 5.6.1` say `use 5.006_001`. This of course won't help: the older Perls won't suddenly start understanding newer features, but at least they will show a sensible error message indicating the required minimum version.

Warning: something's wrong

(W) You passed `warn()` an empty string (the equivalent of `warn ""`) or you called it with no args and `$_` was empty.

Warning: unable to close filehandle %s properly

(S) The implicit `close()` done by an `open()` got an error indication on the `close()`. This usually indicates your file system ran out of disk space.

Warning: Use of "%s" without parentheses is ambiguous

(S ambiguous) You wrote a unary operator followed by something that looks like a binary operator that could also have been interpreted as a term or unary operator. For instance, if you know that the `rand` function has a default argument of 1.0, and you write

```
rand + 5;
```

you may THINK you wrote the same thing as

```
rand() + 5;
```

but in actual fact, you got

```
rand(+5);
```

So put in parentheses to say what you really mean.

Wide character in %s

(W utf8) Perl met a wide character (255) when it wasn't expecting one. This warning is by default on for I/O (like `print`) but can be turned off by `no warnings 'utf8';`. You are supposed to explicitly mark the filehandle with an encoding, see [open](#) and [binmode](#).

`write()` on closed filehandle %s

(W closed) The filehandle you're writing to got itself closed sometime before now. Check your control flow.

X outside of string

(F) You had a pack template that specified a relative position before the beginning of the string being unpacked. See [pack](#).

x outside of string

(F) You had a pack template that specified a relative position after the end of the string being unpacked. See [pack](#).

Xsub "%s" called in sort

(F) The use of an external subroutine as a sort comparison is not yet supported.

Xsub called in sort

(F) The use of an external subroutine as a sort comparison is not yet supported.

YOU HAVEN'T DISABLED SET-ID SCRIPTS IN THE KERNEL YET!

(F) And you probably never will, because you probably don't have the sources to your kernel, and your vendor probably doesn't give a rip about what you want. Your best bet is to put a `setuid` C wrapper around your script.

You need to quote "%s"

(W syntax) You assigned a bareword as a signal handler name. Unfortunately, you already have a subroutine of that name declared, which means that Perl 5 will try to call the subroutine when the assignment is executed, which is probably not what you want. (If it IS what you want, put an `&` in front.)

NAME

perldsc – Perl Data Structures Cookbook

DESCRIPTION

The single feature most sorely lacking in the Perl programming language prior to its 5.0 release was complex data structures. Even without direct language support, some valiant programmers did manage to emulate them, but it was hard work and not for the faint of heart. You could occasionally get away with the `$m{$AoA, $b}` notation borrowed from **awk** in which the keys are actually more like a single concatenated string "`AoAb`", but traversal and sorting were difficult. More desperate programmers even hacked Perl's internal symbol table directly, a strategy that proved hard to develop and maintain—to put it mildly.

The 5.0 release of Perl let us have complex data structures. You may now write something like this and all of a sudden, you'd have an array with three dimensions!

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        for $z (1 .. 10) {
            $AoA[$x][$y][$z] =
                $x ** $y + $z;
        }
    }
}
```

Alas, however simple this may appear, underneath it's a much more elaborate construct than meets the eye!

How do you print it out? Why can't you say just `print @AoA`? How do you sort it? How can you pass it to a function or get one of these back from a function? Is it an object? Can you save it to disk to read back later? How do you access whole rows or columns of that matrix? Do all the values have to be numeric?

As you see, it's quite easy to become confused. While some small portion of the blame for this can be attributed to the reference-based implementation, it's really more due to a lack of existing documentation with examples designed for the beginner.

This document is meant to be a detailed but understandable treatment of the many different sorts of data structures you might want to develop. It should also serve as a cookbook of examples. That way, when you need to create one of these complex data structures, you can just pinch, pilfer, or purloin a drop-in example from here.

Let's look at each of these possible constructs in detail. There are separate sections on each of the following:

- arrays of arrays
- hashes of arrays
- arrays of hashes
- hashes of hashes
- more elaborate constructs

But for now, let's look at general issues common to all these types of data structures.

REFERENCES

The most important thing to understand about all data structures in Perl — including multidimensional arrays—is that even though they might appear otherwise, Perl `@ARRAYs` and `%HASHes` are all internally one-dimensional. They can hold only scalar values (meaning a string, number, or a reference). They cannot directly contain other arrays or hashes, but instead contain *references* to other arrays or hashes.

You can't use a reference to an array or hash in quite the same way that you would a real array or hash. For C or C++ programmers unused to distinguishing between arrays and pointers to the same, this can be confusing. If so, just think of it as the difference between a structure and a pointer to a structure.

You can (and should) read more about references in the perlref(1) man page. Briefly, references are rather like pointers that know what they point to. (Objects are also a kind of reference, but we won't be needing them right away—if ever.) This means that when you have something which looks to you like an access to a two-or-more-dimensional array and/or hash, what's really going on is that the base type is merely a one-dimensional entity that contains references to the next level. It's just that you can *use* it as though it were a two-dimensional one. This is actually the way almost all C multidimensional arrays work as well.

```
$array[7][12]           # array of arrays
$array[7]{string}        # array of hashes
$hash{string}[7]         # hash of arrays
$hash{string}{'another string'} # hash of hashes
```

Now, because the top level contains only references, if you try to print out your array in with a simple `print()` function, you'll get something that doesn't look very nice, like this:

```
@AoA = ( [2, 3], [4, 5, 7], [0] );
print $AoA[1][2];
7
print @AoA;
ARRAY(0x83c38)ARRAY(0x8b194)ARRAY(0x8b1d0)
```

That's because Perl doesn't (ever) implicitly dereference your variables. If you want to get at the thing a reference is referring to, then you have to do this yourself using either prefix typing indicators, like `${$blah}`, `@{$blah}`, `@{$blah[$i]}`, or else postfix pointer arrows, like `$a->[3]`, `$h->{fred}`, or even `$ob->method()->[3]`.

COMMON MISTAKES

The two most common mistakes made in constructing something like an array of arrays is either accidentally counting the number of elements or else taking a reference to the same memory location repeatedly. Here's the case where you just get the count instead of a nested array:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = @array;      # WRONG!
}
```

That's just the simple case of assigning an array to a scalar and getting its element count. If that's what you really and truly want, then you might do well to consider being a tad more explicit about it, like this:

```
for $i (1..10) {
    @array = somefunc($i);
    $counts[$i] = scalar @array;
}
```

Here's the case of taking a reference to the same memory location again and again:

```
for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;      # WRONG!
}
```

So, what's the big problem with that? It looks right, doesn't it? After all, I just told you that you need an array of references, so by golly, you've made me one!

Unfortunately, while this is true, it's still broken. All the references in `@AoA` refer to the *very same place*, and they will therefore all hold whatever was last in `@array`! It's similar to the problem demonstrated in the following C program:

```
#include <pwd.h>
main() {
    struct passwd *getpwnam(), *rp, *dp;
```

```

rp = getpwnam("root");
dp = getpwnam("daemon");

printf("daemon name is %s\nroot name is %s\n",
      dp->pw_name, rp->pw_name);
}

```

Which will print

```

daemon name is daemon
root name is daemon

```

The problem is that both `rp` and `dp` are pointers to the same location in memory! In C, you'd have to remember to `malloc()` yourself some new memory. In Perl, you'll want to use the array constructor `[]` or the hash constructor `{}` instead. Here's the right way to do the preceding broken code fragments:

```

for $i (1..10) {
    @array = somefunc($i);
    $AoA[$i] = [ @array ];
}

```

The square brackets make a reference to a new array with a *copy* of what's in `@array` at the time of the assignment. This is what you want.

Note that this will produce something similar, but it's much harder to read:

```

for $i (1..10) {
    @array = 0 .. $i;
    @{$AoA[$i]} = @array;
}

```

Is it the same? Well, maybe so—and maybe not. The subtle difference is that when you assign something in square brackets, you know for sure it's always a brand new reference with a new *copy* of the data. Something else could be going on in this new case with the `@{$AoA[$i]}` dereference on the left-hand-side of the assignment. It all depends on whether `$AoA[$i]` had been undefined to start with, or whether it already contained a reference. If you had already populated `@AoA` with references, as in

```
$AoA[3] = \@another_array;
```

Then the assignment with the indirection on the left-hand-side would use the existing reference that was already there:

```
@{$AoA[3]} = @array;
```

Of course, this *would* have the "interesting" effect of clobbering `@another_array`. (Have you ever noticed how when a programmer says something is "interesting", that rather than meaning "intriguing", they're disturbingly more apt to mean that it's "annoying", "difficult", or both? :-)

So just remember always to use the array or hash constructors with `[]` or `{}`, and you'll be fine, although it's not always optimally efficient.

Surprisingly, the following dangerous-looking construct will actually work out fine:

```

for $i (1..10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;
}

```

That's because `my()` is more of a run-time statement than it is a compile-time declaration *per se*. This means that the `my()` variable is remade afresh each time through the loop. So even though it *looks* as though you stored the same variable reference each time, you actually did not! This is a subtle distinction that can produce more efficient code at the risk of misleading all but the most experienced of programmers. So I usually advise against teaching it to beginners. In fact, except for passing arguments to functions, I

seldom like to see the gimme-a-reference operator (backslash) used much at all in code. Instead, I advise beginners that they (and most of the rest of us) should try to use the much more easily understood constructors `[]` and `{}` instead of relying upon lexical (or dynamic) scoping and hidden reference-counting to do the right thing behind the scenes.

In summary:

```
$AoA[$i] = [ @array ];      # usually best
$AoA[$i] = \@array;        # perilous; just how my() was that array?
@{ $AoA[$i] } = @array;    # way too tricky for most programmers
```

CAVEAT ON PRECEDENCE

Speaking of things like `@{ $AoA[$i] }`, the following are actually the same thing:

```
$aref->[2][2]      # clear
$$aref[2][2]      # confusing
```

That's because Perl's precedence rules on its five prefix dereferencers (which look like someone swearing: `$`, `@`, `*`, `%`, `&`) make them bind more tightly than the postfix subscripting brackets or braces! This will no doubt come as a great shock to the C or C++ programmer, who is quite accustomed to using `*a[i]` to mean what's pointed to by the *i*'th element of `a`. That is, they first take the subscript, and only then dereference the thing at that subscript. That's fine in C, but this isn't C.

The seemingly equivalent construct in Perl, `$$aref[$i]` first does the deref of `$aref`, making it take `$aref` as a reference to an array, and then dereference that, and finally tell you the *i*'th value of the array pointed to by `$AoA`. If you wanted the C notion, you'd have to write `${ $AoA[$i] }` to force the `$AoA[$i]` to get evaluated first before the leading `$` dereferencer.

WHY YOU SHOULD ALWAYS use strict

If this is starting to sound scarier than it's worth, relax. Perl has some features to help you avoid its most common pitfalls. The best way to avoid getting confused is to start every program like this:

```
#!/usr/bin/perl -w
use strict;
```

This way, you'll be forced to declare all your variables with `my()` and also disallow accidental "symbolic dereferencing". Therefore if you'd done this:

```
my $aref = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $aref[2][2];
```

The compiler would immediately flag that as an error *at compile time*, because you were accidentally accessing `@aref`, an undeclared variable, and it would thereby remind you to write instead:

```
print $aref->[2][2]
```

DEBUGGING

Before version 5.002, the standard Perl debugger didn't do a very nice job of printing out complex data structures. With 5.002 or above, the debugger includes several new features, including command line editing as well as the `x` command to dump out complex data structures. For example, given the assignment to `$AoA` above, here's the debugger output:

```
DB<1> x $AoA
$AoA = ARRAY(0x13b5a0)
0  ARRAY(0x1f0a24)
0  'fred'
```

```

        1  'barney'
        2  'pebbles'
        3  'bambam'
        4  'dino'
1  ARRAY(0x13b558)
    0  'homer'
    1  'bart'
    2  'marge'
    3  'maggie'
2  ARRAY(0x13b540)
    0  'george'
    1  'jane'
    2  'elroy'
    3  'judy'

```

CODE EXAMPLES

Presented with little comment (these will get their own manpages someday) here are short code examples illustrating access of various types of data structures.

ARRAYS OF ARRAYS

Declaration of an ARRAY OF ARRAYS

```

@AoA = (
    [ "fred", "barney" ],
    [ "george", "jane", "elroy" ],
    [ "homer", "marge", "bart" ],
);

```

Generation of an ARRAY OF ARRAYS

```

# reading from file
while ( <> ) {
    push @AoA, [ split ];
}

# calling a function
for $i ( 1 .. 10 ) {
    $AoA[$i] = [ somefunc($i) ];
}

# using temp vars
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $AoA[$i] = [ @tmp ];
}

# add to an existing row
push @{ $AoA[0] }, "wilma", "betty";

```

Access and Printing of an ARRAY OF ARRAYS

```

# one element
$AoA[0][0] = "Fred";

# another element
$AoA[1][1] =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $aref ( @AoA ) {
    print "\t [ @$aref ],\n";
}

```

```

# print the whole thing with indices
for $i ( 0 .. $#AoA ) {
    print "\t [ @{$AoA[$i]} ],\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. ${ $AoA[$i] } ) {
        print "elt $i $j is $AoA[$i][$j]\n";
    }
}

```

HASHES OF ARRAYS

Declaration of a HASH OF ARRAYS

```

%HoA = (
    flintstones      => [ "fred", "barney" ],
    jetsons          => [ "george", "jane", "elroy" ],
    simpsons         => [ "homer", "marge", "bart" ],
);

```

Generation of a HASH OF ARRAYS

```

# reading from file
# flintstones: fred barney wilma dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoA{$1} = [ split ];
}

# reading from file; more temps
# flintstones: fred barney wilma dino
while ( $line = <> ) {
    ($who, $rest) = split /\s*/, $line, 2;
    @fields = split ' ', $rest;
    $HoA{$who} = [ @fields ];
}

# calling a function that returns a list
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoA{$group} = [ get_family($group) ];
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoA{$group} = [ @members ];
}

# append new members to an existing family
push @{$HoA{"flintstones"}}, "wilma", "betty";

```

Access and Printing of a HASH OF ARRAYS

```

# one element
$HoA{flintstones}[0] = "Fred";

# another element
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;

# print the whole thing

```

```

foreach $family ( keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing with indices
foreach $family ( keys %HoA ) {
    print "family: ";
    foreach $i ( 0 .. $# { $HoA{$family} } ) {
        print " $i = $HoA{$family}[$i]";
    }
    print "\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {
    print "$family: @{ $HoA{$family} }\n"
}

# print the whole thing sorted by number of members and name
foreach $family ( sort {
                        @{$HoA{$b}} <=> @{$HoA{$a}}
                        ||
                        $a cmp $b
                    } keys %HoA )
{
    print "$family: ", join(" ", sort @{$HoA{$family}} ), "\n";
}

```

ARRAYS OF HASHES

Declaration of an ARRAY OF HASHES

```

@AoH = (
    {
        Lead    => "fred",
        Friend   => "barney",
    },
    {
        Lead    => "george",
        Wife     => "jane",
        Son      => "elroy",
    },
    {
        Lead    => "homer",
        Wife     => "marge",
        Son      => "bart",
    }
);

```

Generation of an ARRAY OF HASHES

```

# reading from file
# format: LEAD=fred FRIEND=barney
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}

```

```

    push @AoH, $rec;
}

# reading from file
# format: LEAD=fred FRIEND=barney
# no temp
while ( <> ) {
    push @AoH, { split /\s+=/ };
}

# calling a function that returns a key/value pair list, like
# "lead","fred","daughter","pebbles"
while ( %fields = getnextpairset() ) {
    push @AoH, { %fields };
}

# likewise, but using no temp vars
while (<>) {
    push @AoH, { parsepairs($_) };
}

# add key/value to an element
$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";

```

Access and Printing of an ARRAY OF HASHES

```

# one element
$AoH[0]{lead} = "fred";

# another element
$AoH[1]{lead} =~ s/(\w)/\u$1/;

# print the whole thing with refs
for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}

# print the whole thing with indices
for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
        print "$role=$AoH[$i]{$role} ";
    }
    print "}\n";
}

# print the whole thing one at a time
for $i ( 0 .. $#AoH ) {
    for $role ( keys %{ $AoH[$i] } ) {
        print "elt $i $role is $AoH[$i]{$role}\n";
    }
}

```


HASHES OF HASHES

Declaration of a HASH OF HASHES

```
%HoH = (
    flintstones => {
        lead      => "fred",
        pal       => "barney",
    },
    jetsons      => {
        lead      => "george",
        wife      => "jane",
        "his boy" => "elroy",
    },
    simpsons     => {
        lead      => "homer",
        wife      => "marge",
        kid       => "bart",
    },
);
```

Generation of a HASH OF HASHES

```
# reading from file
# flintstones: lead=fred pal=barney wife=wilma pet=dino
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $HoH{$who}{$key} = $value;
    }
}

# reading from file; more temps
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $who = $1;
    $rec = {};
    $HoH{$who} = $rec;
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
}

# calling a function that returns a key,value hash
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

# likewise, but using temps
for $group ( "simpsons", "jetsons", "flintstones" ) {
    %members = get_family($group);
    $HoH{$group} = { %members };
}

# append new members to an existing family
%new_folks = (
    wife => "wilma",
```

```

    pet => "dino",
);

for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}

```

Access and Printing of a HASH OF HASHES

```

# one element
$HoH{flintstones}{wife} = "wilma";

# another element
$HoH{simpsons}{lead} =~ s/(\w)/\u$1/;

# print the whole thing
foreach $family ( keys %HoH ) {
    print "$family: { ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# print the whole thing somewhat sorted
foreach $family ( sort keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# print the whole thing sorted by number of members
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } } keys %HoH ) {
    print "$family: { ";
    for $role ( sort keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

# establish a sort order (rank) for each role
$i = 0;
for ( qw(lead wife son daughter pal pet) ) { $rank{$_} = ++$i }

# now print the whole thing sorted by number of members
foreach $family ( sort { keys %{ $HoH{$b} } <=> keys %{ $HoH{$a} } } keys %HoH ) {
    print "$family: { ";
    # and print these according to rank order
    for $role ( sort { $rank{$a} <=> $rank{$b} } keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "}\n";
}

```

MORE ELABORATE RECORDS

Declaration of MORE ELABORATE RECORDS

Here's a sample showing how to create and use a record whose fields are of many different sorts:

```

$rec = {
    TEXT      => $string,
    SEQUENCE  => [ @old_values ],
    LOOKUP    => { %some_table },
    THATCODE  => \&some_function,
    THISCODE  => sub { $_[0] ** $_[1] },
    HANDLE    => \*STDOUT,
};

print $rec->{TEXT};

print $rec->{SEQUENCE}[0];
$last = pop @ { $rec->{SEQUENCE} };

print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{ $rec->{LOOKUP} };

$answer = $rec->{THATCODE}->($arg);
$answer = $rec->{THISCODE}->($arg1, $arg2);

# careful of extra block braces on fh ref
print { $rec->{HANDLE} } "a string\n";

use FileHandle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print(" a string\n");

```

Declaration of a HASH OF COMPLEX RECORDS

```

%TV = (
    flintstones => {
        series    => "flintstones",
        nights    => [ qw(monday thursday friday) ],
        members   => [
            { name => "fred",    role => "lead", age  => 36, },
            { name => "wilma",   role => "wife", age  => 31, },
            { name => "pebbles", role => "kid",  age  => 4,  },
        ],
    },
    jetsons      => {
        series    => "jetsons",
        nights    => [ qw(wednesday saturday) ],
        members   => [
            { name => "george",  role => "lead", age  => 41, },
            { name => "jane",    role => "wife", age  => 39, },
            { name => "elroy",   role => "kid",  age  => 9,  },
        ],
    },
    simpsons     => {
        series    => "simpsons",
        nights    => [ qw(monday) ],
        members   => [
            { name => "homer",   role => "lead", age  => 34, },
            { name => "marge",   role => "wife", age  => 37, },
            { name => "bart",    role => "kid",  age  => 11, },
        ],
    },
);

```

Generation of a HASH OF COMPLEX RECORDS

```

# reading from file
# this is most easily done by having the file itself be
# in the raw data format as shown above. perl is happy
# to parse complex data structures if declared as data, so
# sometimes it's easiest to do that

# here's a piece by piece build up
$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days() ];

@members = ();
# assume this file in field=value syntax
while (<>) {
    %fields = split /\s=/;
    push @members, { %fields };
}
$rec->{members} = [ @members ];

# now remember the whole thing
$TV{ $rec->{series} } = $rec;

#####
# now, you might want to make interesting extra fields that
# include pointers back into the same data structure so if
# change one piece, it changes everywhere, like for example
# if you wanted a {kids} field that was a reference
# to an array of the kids' records without having duplicate
# records and thus update problems.
#####
foreach $family (keys %TV) {
    $rec = $TV{$family}; # temp pointer
    @kids = ();
    for $person ( @{ $rec->{members} } ) {
        if ($person->{role} =~ /kid|son|daughter/) {
            push @kids, $person;
        }
    }
    # REMEMBER: $rec and $TV{$family} point to same data!!
    $rec->{kids} = [ @kids ];
}

# you copied the array, but the array itself contains pointers
# to uncopied objects. this means that if you make bart get
# older via

$TV{simpsons}{kids}[0]{age}++;

# then this would also change in
print $TV{simpsons}{members}[2]{age};

# because $TV{simpsons}{kids}[0] and $TV{simpsons}{members}[2]
# both point to the same underlying anonymous hash table

# print the whole thing
foreach $family ( keys %TV ) {
    print "the $family";
}

```

```
print " is on during @{ $TV{$family}{nights} }\n";
print "its members are:\n";
for $who ( @{ $TV{$family}{members} } ) {
    print " $who->{name} ($who->{role}), age $who->{age}\n";
}
print "it turns out that $TV{$family}{lead} has ";
print scalar ( @{ $TV{$family}{kids} } ), " kids named ";
print join (", ", map { $_->{name} } @{ $TV{$family}{kids} } );
print "\n";
}
```

Database Ties

You cannot easily tie a multilevel data structure (such as a hash of hashes) to a dbm file. The first problem is that all but GDBM and Berkeley DB have size limitations, but beyond that, you also have problems with how references are to be represented on disk. One experimental module that does partially attempt to address this need is the MLDBM module. Check your nearest CPAN site as described in [perlmodlib](#) for source code to MLDBM.

SEE ALSO

perlref(1), perllob(1), perldata(1), perlobj(1)

AUTHOR

Tom Christiansen <tchrist@perl.com>

Last update: Wed Oct 23 04:57:50 MET DST 1996

NAME

perlebcdic – Considerations for running Perl on EBCDIC platforms

DESCRIPTION

An exploration of some of the issues facing Perl programmers on EBCDIC based computers. We do not cover localization, internationalization, or multi byte character set issues other than some discussion of UTF-8 and UTF-EBCDIC.

Portions that are still incomplete are marked with XXX.

COMMON CHARACTER CODE SETS**ASCII**

The American Standard Code for Information Interchange is a set of integers running from 0 to 127 (decimal) that imply character interpretation by the display and other system(s) of computers. The range 0..127 can be covered by setting the bits in a 7-bit binary digit, hence the set is sometimes referred to as a "7-bit ASCII". ASCII was described by the American National Standards Institute document ANSI X3.4-1986. It was also described by ISO 646:1991 (with localization for currency symbols). The full ASCII set is given in the table below as the first 128 elements. Languages that can be written adequately with the characters in ASCII include English, Hawaiian, Indonesian, Swahili and some Native American languages.

There are many character sets that extend the range of integers from $0..2^{**7}-1$ up to $2^{**8}-1$, or 8 bit bytes (octets if you prefer). One common one is the ISO 8859-1 character set.

ISO 8859

The ISO 8859-*\$n* are a collection of character code sets from the International Organization for Standardization (ISO) each of which adds characters to the ASCII set that are typically found in European languages many of which are based on the Roman, or Latin, alphabet.

Latin 1 (ISO 8859-1)

A particular 8-bit extension to ASCII that includes grave and acute accented Latin characters. Languages that can employ ISO 8859-1 include all the languages covered by ASCII as well as Afrikaans, Albanian, Basque, Catalan, Danish, Faroese, Finnish, Norwegian, Portuguese, Spanish, and Swedish. Dutch is covered albeit without the ij ligature. French is covered too but without the oe ligature. German can use ISO 8859-1 but must do so without German-style quotation marks. This set is based on Western European extensions to ASCII and is commonly encountered in world wide web work. In IBM character code set identification terminology ISO 8859-1 is also known as CCSID 819 (or sometimes 0819 or even 00819).

EBCDIC

The Extended Binary Coded Decimal Interchange Code refers to a large collection of slightly different single and multi byte coded character sets that are different from ASCII or ISO 8859-1 and typically run on host computers. The EBCDIC encodings derive from 8 bit byte extensions of Hollerith punched card encodings. The layout on the cards was such that high bits were set for the upper and lower case alphabet characters [a-z] and [A-Z], but there were gaps within each latin alphabet range.

Some IBM EBCDIC character sets may be known by character code set identification numbers (CCSID numbers) or code page numbers. Leading zero digits in CCSID numbers within this document are insignificant. E.g. CCSID 0037 may be referred to as 37 in places.

13 variant characters

Among IBM EBCDIC character code sets there are 13 characters that are often mapped to different integer values. Those characters are known as the 13 "variant" characters and are:

`\ [] { } ^ ~ ! # | $ @ ``

0037

Character code set ID 0037 is a mapping of the ASCII plus Latin-1 characters (i.e. ISO 8859-1) to an EBCDIC set. 0037 is used in North American English locales on the OS/400 operating system that runs on AS/400 computers. CCSID 37 differs from ISO 8859-1 in 237 places, in other words they agree on only 19 code point values.

1047

Character code set ID 1047 is also a mapping of the ASCII plus Latin-1 characters (i.e. ISO 8859-1) to an EBCDIC set. 1047 is used under Unix System Services for OS/390 or z/OS, and OpenEdition for VM/ESA. CCSID 1047 differs from CCSID 0037 in eight places.

POSIX-BC

The EBCDIC code page in use on Siemens' BS2000 system is distinct from 1047 and 0037. It is identified below as the POSIX-BC set.

Unicode code points versus EBCDIC code points

In Unicode terminology a *code point* is the number assigned to a character: for example, in EBCDIC the character "A" is usually assigned the number 193. In Unicode the character "A" is assigned the number 65. This causes a problem with the semantics of the pack/unpack "U", which are supposed to pack Unicode code points to characters and back to numbers. The problem is: which code points to use for code points less than 256? (for 256 and over there's no problem: Unicode code points are used) In EBCDIC, for the low 256 the EBCDIC code points are used. This means that the equivalences

```
pack("U", ord($character)) eq $character
unpack("U", $character) == ord $character
```

will hold. (If Unicode code points were applied consistently over all the possible code points, pack("U",ord("A")) would in EBCDIC equal *A with acute* or chr(101), and unpack("U", "A") would equal 65, or *non-breaking space*, not 193, or ord "A".)

Remaining Perl Unicode problems in EBCDIC

- Many of the remaining seem to be related to case-insensitive matching: for example, `< / [\x{131}] /` (LATIN SMALL LETTER DOTLESS I) does not match "I" case-insensitively, as it should under Unicode. (The match succeeds in ASCII-derived platforms.)
- The extensions `Unicode::Collate` and `Unicode::Normalized` are not supported under EBCDIC, likewise for the encoding pragma.

Unicode and UTF

UTF is a Unicode Transformation Format. UTF-8 is a Unicode conforming representation of the Unicode standard that looks very much like ASCII. UTF-EBCDIC is an attempt to represent Unicode characters in an EBCDIC transparent manner.

Using Encode

Starting from Perl 5.8 you can use the standard new module Encode to translate from EBCDIC to Latin-1 code points

```
use Encode 'from_to';
my %ebcdic = ( 176 => 'cp37', 95 => 'cp1047', 106 => 'posix-bc' );
# $a is in EBCDIC code points
from_to($a, %ebcdic{ord '^'}, 'latin1');
# $a is ISO 8859-1 code points
```

and from Latin-1 code points to EBCDIC code points

```
use Encode 'from_to';
```

```
my %ebcdic = ( 176 => 'cp37', 95 => 'cp1047', 106 => 'posix-bc' );
# $a is ISO 8859-1 code points
from_to($a, 'latin1', %ebcdic{ord '^'});
# $a is in EBCDIC code points
```

For doing I/O it is suggested that you use the autotranslating features of PerlIO, see [perluniintro](#).

SINGLE OCTET TABLES

The following tables list the ASCII and Latin 1 ordered sets including the subsets: C0 controls (0..31), ASCII graphics (32..7e), delete (7f), C1 controls (80..9f), and Latin-1 (a.k.a. ISO 8859-1) (a0..ff). In the table non-printing control character names as well as the Latin 1 extensions to ASCII have been labelled with character names roughly corresponding to *The Unicode Standard, Version 3.0* albeit with substitutions such as s/LATIN// and s/VULGAR// in all cases, s/CAPITAL LETTER// in some cases, and s/SMALL LETTER ([A-Z])|l\$1/ in some other cases (the charnames pragma names unfortunately do not list explicit names for the C0 or C1 control characters). The "names" of the C1 control set (128..159 in ISO 8859-1) listed here are somewhat arbitrary. The differences between the 0037 and 1047 sets are flagged with ***. The differences between the 1047 and POSIX-BC sets are flagged with ###. All ord() numbers listed are decimal. If you would rather see this table listing octal values then run the table (that is, the pod version of this document since this recipe may not work with a pod2_ other_ format translation) through:

recipe 0

```
perl -ne 'if (/(.{33}) (\d+)\s+(\d+)\s+(\d+)\s+(\d+)/)' \
-e '{printf("%s%-9%-9%-9%o\n", $1, $2, $3, $4, $5)}' perlebcdic.pod
```

If you want to retain the UTF-x code points then in script form you might want to write:

recipe 1

```
open(FH,"<perlebcdic.pod") or die "Could not open perlebcdic.pod: $!";
while (<FH>) {
    if (/(.{33})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+\.?(\d*)\s+(\d+)\s+\.?(\d*)/)
        if ($7 ne '' && $9 ne '') {
            printf("%s%-9o%-9o%-9o%-9o%-3o.%-5o%-3o.o\n",$1,$2,$3,$4,$5,$6,$7,$8)
        }
        elsif ($7 ne '') {
            printf("%s%-9o%-9o%-9o%-9o%-3o.%-5oo\n",$1,$2,$3,$4,$5,$6,$7,$8);
        }
        else {
            printf("%s%-9o%-9o%-9o%-9o%-9oo\n",$1,$2,$3,$4,$5,$6,$8);
        }
    }
}
```

If you would rather see this table listing hexadecimal values then run the table through:

recipe 2

```
perl -ne 'if (/(.{33})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)/)' \
-e '{printf("%s%-9X%-9X%-9X\n",$1,$2,$3,$4,$5)}' perlebcdic.pod
```

Or, in order to retain the UTF-x code points in hexadecimal:

recipe 3

```
open(FH,"<perlebcdic.pod") or die "Could not open perlebcdic.pod: $!";
while (<FH>) {
    if (/(.{33})(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+(\d+)\s+\.?(\d*)\s+(\d+)\s+\.?(\d*)/)
        if ($7 ne '' && $9 ne '') {
            printf("%s%-9X%-9X%-9X%-9X%-2X.%-6X%-2X.%X\n",$1,$2,$3,$4,$5,$6,$7,$8
```



```

    }
    elsif ($7 ne '') {
        printf("%s%-9X%-9X%-9X%-9X%-2X.%-6X%X\n", $1, $2, $3, $4, $5, $6, $7, $8);
    }
    else {
        printf("%s%-9X%-9X%-9X%-9X%-9X%X\n", $1, $2, $3, $4, $5, $6, $8);
    }
}
}

```

chr	8859-1		1047	POSIX-BC	incomp- lete	incomp- lete
	0819	0037			UTF-8	UTF-EBC
<NULL>	0	0	0	0	0	0
<START OF HEADING>	1	1	1	1	1	1
<START OF TEXT>	2	2	2	2	2	2
<END OF TEXT>	3	3	3	3	3	3
<END OF TRANSMISSION>	4	55	55	55	4	55
<ENQUIRY>	5	45	45	45	5	45
<ACKNOWLEDGE>	6	46	46	46	6	46
<BELL>	7	47	47	47	7	47
<BACKSPACE>	8	22	22	22	8	22
<HORIZONTAL TABULATION>	9	5	5	5	9	5
<LINE FEED>	10	37	21	21	10	21
<VERTICAL TABULATION>	11	11	11	11	11	11
<FORM FEED>	12	12	12	12	12	12
<CARRIAGE RETURN>	13	13	13	13	13	13
<SHIFT OUT>	14	14	14	14	14	14
<SHIFT IN>	15	15	15	15	15	15
<DATA LINK ESCAPE>	16	16	16	16	16	16
<DEVICE CONTROL ONE>	17	17	17	17	17	17
<DEVICE CONTROL TWO>	18	18	18	18	18	18
<DEVICE CONTROL THREE>	19	19	19	19	19	19
<DEVICE CONTROL FOUR>	20	60	60	60	20	60
<NEGATIVE ACKNOWLEDGE>	21	61	61	61	21	61
<SYNCHRONOUS IDLE>	22	50	50	50	22	50
<END OF TRANSMISSION BLOCK>	23	38	38	38	23	38
<CANCEL>	24	24	24	24	24	24
<END OF MEDIUM>	25	25	25	25	25	25
<SUBSTITUTE>	26	63	63	63	26	63
<ESCAPE>	27	39	39	39	27	39
<FILE SEPARATOR>	28	28	28	28	28	28
<GROUP SEPARATOR>	29	29	29	29	29	29
<RECORD SEPARATOR>	30	30	30	30	30	30
<UNIT SEPARATOR>	31	31	31	31	31	31
<SPACE>	32	64	64	64	32	64
!	33	90	90	90	33	90
"	34	127	127	127	34	127
#	35	123	123	123	35	123
\$	36	91	91	91	36	91
%	37	108	108	108	37	108
&	38	80	80	80	38	80
'	39	125	125	125	39	125
(40	77	77	77	40	77

)	41	93	93	93	41	93
*	42	92	92	92	42	92
+	43	78	78	78	43	78
,	44	107	107	107	44	107
-	45	96	96	96	45	96
.	46	75	75	75	46	75
/	47	97	97	97	47	97
0	48	240	240	240	48	240
1	49	241	241	241	49	241
2	50	242	242	242	50	242
3	51	243	243	243	51	243
4	52	244	244	244	52	244
5	53	245	245	245	53	245
6	54	246	246	246	54	246
7	55	247	247	247	55	247
8	56	248	248	248	56	248
9	57	249	249	249	57	249
:	58	122	122	122	58	122
;	59	94	94	94	59	94
<	60	76	76	76	60	76
=	61	126	126	126	61	126
>	62	110	110	110	62	110
?	63	111	111	111	63	111
@	64	124	124	124	64	124
A	65	193	193	193	65	193
B	66	194	194	194	66	194
C	67	195	195	195	67	195
D	68	196	196	196	68	196
E	69	197	197	197	69	197
F	70	198	198	198	70	198
G	71	199	199	199	71	199
H	72	200	200	200	72	200
I	73	201	201	201	73	201
J	74	209	209	209	74	209
K	75	210	210	210	75	210
L	76	211	211	211	76	211
M	77	212	212	212	77	212
N	78	213	213	213	78	213
O	79	214	214	214	79	214
P	80	215	215	215	80	215
Q	81	216	216	216	81	216
R	82	217	217	217	82	217
S	83	226	226	226	83	226
T	84	227	227	227	84	227
U	85	228	228	228	85	228
V	86	229	229	229	86	229
W	87	230	230	230	87	230
X	88	231	231	231	88	231
Y	89	232	232	232	89	232
Z	90	233	233	233	90	233
[91	186	173	187	91	173
\	92	224	224	188	92	224
]	93	187	189	189	93	189
^	94	176	95	106	94	95

—	95	109	109	109	95	109
`	96	121	121	74	96	121
a	97	129	129	129	97	129
b	98	130	130	130	98	130
c	99	131	131	131	99	131
d	100	132	132	132	100	132
e	101	133	133	133	101	133
f	102	134	134	134	102	134
g	103	135	135	135	103	135
h	104	136	136	136	104	136
i	105	137	137	137	105	137
j	106	145	145	145	106	145
k	107	146	146	146	107	146
l	108	147	147	147	108	147
m	109	148	148	148	109	148
n	110	149	149	149	110	149
o	111	150	150	150	111	150
p	112	151	151	151	112	151
q	113	152	152	152	113	152
r	114	153	153	153	114	153
s	115	162	162	162	115	162
t	116	163	163	163	116	163
u	117	164	164	164	117	164
v	118	165	165	165	118	165
w	119	166	166	166	119	166
x	120	167	167	167	120	167
y	121	168	168	168	121	168
z	122	169	169	169	122	169
{	123	192	192	251	123	192
	124	79	79	79	124	79
}	125	208	208	253	125	208
~	126	161	161	255	126	161
<DELETE>	127	7	7	7	127	7
<C1 0>	128	32	32	32	194.128	32
<C1 1>	129	33	33	33	194.129	33
<C1 2>	130	34	34	34	194.130	34
<C1 3>	131	35	35	35	194.131	35
<C1 4>	132	36	36	36	194.132	36
<C1 5>	133	21	37	37	194.133	37
<C1 6>	134	6	6	6	194.134	6
<C1 7>	135	23	23	23	194.135	23
<C1 8>	136	40	40	40	194.136	40
<C1 9>	137	41	41	41	194.137	41
<C1 10>	138	42	42	42	194.138	42
<C1 11>	139	43	43	43	194.139	43
<C1 12>	140	44	44	44	194.140	44
<C1 13>	141	9	9	9	194.141	9
<C1 14>	142	10	10	10	194.142	10
<C1 15>	143	27	27	27	194.143	27
<C1 16>	144	48	48	48	194.144	48
<C1 17>	145	49	49	49	194.145	49
<C1 18>	146	26	26	26	194.146	26
<C1 19>	147	51	51	51	194.147	51
<C1 20>	148	52	52	52	194.148	52

<C1 21>	149	53	53	53	194.149	53
<C1 22>	150	54	54	54	194.150	54
<C1 23>	151	8	8	8	194.151	8
<C1 24>	152	56	56	56	194.152	56
<C1 25>	153	57	57	57	194.153	57
<C1 26>	154	58	58	58	194.154	58
<C1 27>	155	59	59	59	194.155	59
<C1 28>	156	4	4	4	194.156	4
<C1 29>	157	20	20	20	194.157	20
<C1 30>	158	62	62	62	194.158	62
<C1 31>	159	255	255	95	194.159	255
<NON-BREAKING SPACE>	160	65	65	65	194.160	128.65
<INVERTED EXCLAMATION MARK>	161	170	170	170	194.161	128.66
<CENT SIGN>	162	74	74	176	194.162	128.67
<POUND SIGN>	163	177	177	177	194.163	128.68
<CURRENCY SIGN>	164	159	159	159	194.164	128.69
<YEN SIGN>	165	178	178	178	194.165	128.70
<BROKEN BAR>	166	106	106	208	194.166	128.71
<SECTION SIGN>	167	181	181	181	194.167	128.72
<DIAERESIS>	168	189	187	121	194.168	128.73
<COPYRIGHT SIGN>	169	180	180	180	194.169	128.74
<FEMININE ORDINAL INDICATOR>	170	154	154	154	194.170	128.81
<LEFT POINTING GUILLEMET>	171	138	138	138	194.171	128.82
<NOT SIGN>	172	95	176	186	194.172	128.83
<SOFT HYPHEN>	173	202	202	202	194.173	128.84
<REGISTERED TRADE MARK SIGN>	174	175	175	175	194.174	128.85
<MACRON>	175	188	188	161	194.175	128.86
<DEGREE SIGN>	176	144	144	144	194.176	128.87
<PLUS-OR-MINUS SIGN>	177	143	143	143	194.177	128.88
<SUPERSCRIT TWO>	178	234	234	234	194.178	128.89
<SUPERSCRIT THREE>	179	250	250	250	194.179	128.98
<ACUTE ACCENT>	180	190	190	190	194.180	128.99
<MICRO SIGN>	181	160	160	160	194.181	128.100
<PARAGRAPH SIGN>	182	182	182	182	194.182	128.101
<MIDDLE DOT>	183	179	179	179	194.183	128.102
<CEDILLA>	184	157	157	157	194.184	128.103
<SUPERSCRIT ONE>	185	218	218	218	194.185	128.104
<MASC. ORDINAL INDICATOR>	186	155	155	155	194.186	128.105
<RIGHT POINTING GUILLEMET>	187	139	139	139	194.187	128.106
<FRACTION ONE QUARTER>	188	183	183	183	194.188	128.112
<FRACTION ONE HALF>	189	184	184	184	194.189	128.113
<FRACTION THREE QUARTERS>	190	185	185	185	194.190	128.114
<INVERTED QUESTION MARK>	191	171	171	171	194.191	128.115
<A WITH GRAVE>	192	100	100	100	195.128	138.65
<A WITH ACUTE>	193	101	101	101	195.129	138.66
<A WITH CIRCUMFLEX>	194	98	98	98	195.130	138.67
<A WITH TILDE>	195	102	102	102	195.131	138.68
<A WITH DIAERESIS>	196	99	99	99	195.132	138.69
<A WITH RING ABOVE>	197	103	103	103	195.133	138.70
<CAPITAL LIGATURE AE>	198	158	158	158	195.134	138.71
<C WITH CEDILLA>	199	104	104	104	195.135	138.72
<E WITH GRAVE>	200	116	116	116	195.136	138.73
<E WITH ACUTE>	201	113	113	113	195.137	138.74
<E WITH CIRCUMFLEX>	202	114	114	114	195.138	138.81

<E WITH DIAERESIS>	203	115	115	115	195.139	138.82
<I WITH GRAVE>	204	120	120	120	195.140	138.83
<I WITH ACUTE>	205	117	117	117	195.141	138.84
<I WITH CIRCUMFLEX>	206	118	118	118	195.142	138.85
<I WITH DIAERESIS>	207	119	119	119	195.143	138.86
<CAPITAL LETTER ETH>	208	172	172	172	195.144	138.87
<N WITH TILDE>	209	105	105	105	195.145	138.88
<O WITH GRAVE>	210	237	237	237	195.146	138.89
<O WITH ACUTE>	211	238	238	238	195.147	138.98
<O WITH CIRCUMFLEX>	212	235	235	235	195.148	138.99
<O WITH TILDE>	213	239	239	239	195.149	138.100
<O WITH DIAERESIS>	214	236	236	236	195.150	138.101
<MULTIPLICATION SIGN>	215	191	191	191	195.151	138.102
<O WITH STROKE>	216	128	128	128	195.152	138.103
<U WITH GRAVE>	217	253	253	224	195.153	138.104
<U WITH ACUTE>	218	254	254	254	195.154	138.105
<U WITH CIRCUMFLEX>	219	251	251	221	195.155	138.106
<U WITH DIAERESIS>	220	252	252	252	195.156	138.112
<Y WITH ACUTE>	221	173	186	173	195.157	138.113
<CAPITAL LETTER THORN>	222	174	174	174	195.158	138.114
<SMALL LETTER SHARP S>	223	89	89	89	195.159	138.115
<a WITH GRAVE>	224	68	68	68	195.160	139.65
<a WITH ACUTE>	225	69	69	69	195.161	139.66
<a WITH CIRCUMFLEX>	226	66	66	66	195.162	139.67
<a WITH TILDE>	227	70	70	70	195.163	139.68
<a WITH DIAERESIS>	228	67	67	67	195.164	139.69
<a WITH RING ABOVE>	229	71	71	71	195.165	139.70
<SMALL LIGATURE ae>	230	156	156	156	195.166	139.71
<c WITH CEDILLA>	231	72	72	72	195.167	139.72
<e WITH GRAVE>	232	84	84	84	195.168	139.73
<e WITH ACUTE>	233	81	81	81	195.169	139.74
<e WITH CIRCUMFLEX>	234	82	82	82	195.170	139.81
<e WITH DIAERESIS>	235	83	83	83	195.171	139.82
<i WITH GRAVE>	236	88	88	88	195.172	139.83
<i WITH ACUTE>	237	85	85	85	195.173	139.84
<i WITH CIRCUMFLEX>	238	86	86	86	195.174	139.85
<i WITH DIAERESIS>	239	87	87	87	195.175	139.86
<SMALL LETTER eth>	240	140	140	140	195.176	139.87
<n WITH TILDE>	241	73	73	73	195.177	139.88
<o WITH GRAVE>	242	205	205	205	195.178	139.89
<o WITH ACUTE>	243	206	206	206	195.179	139.98
<o WITH CIRCUMFLEX>	244	203	203	203	195.180	139.99
<o WITH TILDE>	245	207	207	207	195.181	139.100
<o WITH DIAERESIS>	246	204	204	204	195.182	139.101
<DIVISION SIGN>	247	225	225	225	195.183	139.102
<o WITH STROKE>	248	112	112	112	195.184	139.103
<u WITH GRAVE>	249	221	221	192	195.185	139.104
<u WITH ACUTE>	250	222	222	222	195.186	139.105
<u WITH CIRCUMFLEX>	251	219	219	219	195.187	139.106
<u WITH DIAERESIS>	252	220	220	220	195.188	139.112
<y WITH ACUTE>	253	141	141	141	195.189	139.113
<SMALL LETTER thorn>	254	142	142	142	195.190	139.114
<y WITH DIAERESIS>	255	223	223	223	195.191	139.115

If you would rather see the above table in CCSID 0037 order rather than ASCII + Latin-1 order then run the table through:

recipe 4

```
perl -ne 'if(/.{33}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}/)' \
-e '{push(@l,$_)}' \
-e 'END{print map{$_->[0]}' \
-e '      sort{$a->[1] <=> $b->[1]}' \
-e '      map{[$_,substr($_,42,3)]}@l;}' perlebcdic.pod
```

If you would rather see it in CCSID 1047 order then change the digit 42 in the last line to 51, like this:

recipe 5

```
perl -ne 'if(/.{33}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}/)' \
-e '{push(@l,$_)}' \
-e 'END{print map{$_->[0]}' \
-e '      sort{$a->[1] <=> $b->[1]}' \
-e '      map{[$_,substr($_,51,3)]}@l;}' perlebcdic.pod
```

If you would rather see it in POSIX-BC order then change the digit 51 in the last line to 60, like this:

recipe 6

```
perl -ne 'if(/.{33}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}\s{6,8}\d{1,3}/)' \
-e '{push(@l,$_)}' \
-e 'END{print map{$_->[0]}' \
-e '      sort{$a->[1] <=> $b->[1]}' \
-e '      map{[$_,substr($_,60,3)]}@l;}' perlebcdic.pod
```

IDENTIFYING CHARACTER CODE SETS

To determine the character set you are running under from perl one could use the return value of `ord()` or `chr()` to test one or more character values. For example:

```
$is_ascii = "A" eq chr(65);
$is_ebcdic = "A" eq chr(193);
```

Also, `"\t"` is a HORIZONTAL TABULATION character so that:

```
$is_ascii = ord("\t") == 9;
$is_ebcdic = ord("\t") == 5;
```

To distinguish EBCDIC code pages try looking at one or more of the characters that differ between them. For example:

```
$is_ebcdic_37 = "\n" eq chr(37);
$is_ebcdic_1047 = "\n" eq chr(21);
```

Or better still choose a character that is uniquely encoded in any of the code sets, e.g.:

```
$is_ascii = ord('[') == 91;
$is_ebcdic_37 = ord('[') == 186;
$is_ebcdic_1047 = ord('[') == 173;
$is_ebcdic_POSIX_BC = ord('[') == 187;
```

However, it would be unwise to write tests such as:

```
$is_ascii = "\r" ne chr(13); # WRONG
$is_ascii = "\n" ne chr(10); # ILL ADVISED
```

Obviously the first of these will fail to distinguish most ASCII machines from either a CCSID 0037, a 1047, or a POSIX-BC EBCDIC machine since `"\r"` eq `chr(13)` under all of those coded character sets. But note

too that because "\n" is chr(13) and "\r" is chr(10) on the MacIntosh (which is an ASCII machine) the second `$is_ascii` test will lead to trouble there.

To determine whether or not perl was built under an EBCDIC code page you can use the Config module like so:

```
use Config;
$is_ebcdic = $Config{'ebcdic'} eq 'define';
```

CONVERSIONS

tr///

In order to convert a string of characters from one character set to another a simple list of numbers, such as in the right columns in the above table, along with perl's `tr///` operator is all that is needed. The data in the table are in ASCII order hence the EBCDIC columns provide easy to use ASCII to EBCDIC operations that are also easily reversed.

For example, to convert ASCII to code page 037 take the output of the second column from the output of recipe 0 (modified to add \\ characters) and use it in `tr///` like so:

```
$cp_037 =
'\000\001\002\003\234\011\206\177\227\215\216\013\014\015\016\017' .
'\020\021\022\023\235\205\010\207\030\031\222\217\034\035\036\037' .
'\200\201\202\203\204\012\027\033\210\211\212\213\214\005\006\007' .
'\220\221\026\223\224\225\226\004\230\231\232\233\024\025\236\032' .
'\040\240\342\344\340\341\343\345\347\361\242\056\074\050\053\174' .
'\046\351\352\353\350\355\356\357\354\337\041\044\052\051\073\254' .
'\055\057\302\304\300\301\303\305\307\321\246\054\045\137\076\077' .
'\370\311\312\313\310\315\316\317\314\140\072\043\100\047\075\042' .
'\330\141\142\143\144\145\146\147\150\151\253\273\360\375\376\261' .
'\260\152\153\154\155\156\157\160\161\162\252\272\346\270\306\244' .
'\265\176\163\164\165\166\167\170\171\172\241\277\320\335\336\256' .
'\136\243\245\267\251\247\266\274\275\276\133\135\257\250\264\327' .
'\173\101\102\103\104\105\106\107\110\111\255\364\366\362\363\365' .
'\175\112\113\114\115\116\117\120\121\122\271\373\374\371\372\377' .
'\134\367\123\124\125\126\127\130\131\132\262\324\326\322\323\325' .
'\060\061\062\063\064\065\066\067\070\071\263\333\334\331\332\237' ;

my $ebcdic_string = $ascii_string;
eval '$ebcdic_string =~ tr/\000-\377/' . $cp_037 . '/';
```

To convert from EBCDIC 037 to ASCII just reverse the order of the `tr///` arguments like so:

```
my $ascii_string = $ebcdic_string;
eval '$ascii_string = tr/' . $cp_037 . '/\000-\377/' ;
```

Similarly one could take the output of the third column from recipe 0 to obtain a `$cp_1047` table. The fourth column of the output from recipe 0 could provide a `$cp_posix_bc` table suitable for transcoding as well.

iconv

XPG operability often implies the presence of an *iconv* utility available from the shell or from the C library. Consult your system's documentation for information on *iconv*.

On OS/390 or z/OS see the *iconv(1)* manpage. One way to invoke the *iconv* shell utility from within perl would be to:

```
# OS/390 or z/OS example
$ascii_data = `echo '$ebcdic_data' | iconv -f IBM-1047 -t ISO8859-1`
```

or the inverse map:

```
# OS/390 or z/OS example
$ebcdic_data = `echo '$ascii_data' | iconv -f ISO8859-1 -t IBM-1047`
```

For other perl based conversion options see the `Convert::*` modules on CPAN.

C RTL

The OS/390 and z/OS C run time libraries provide `_atoe()` and `_etoa()` functions.

OPERATOR DIFFERENCES

The `..` range operator treats certain character ranges with care on EBCDIC machines. For example the following array will have twenty six elements on either an EBCDIC machine or an ASCII machine:

```
@alphabet = ('A'..'Z'); # $#alphabet == 25
```

The bitwise operators such as `&` `^` `|` may return different results when operating on string or character data in a perl program running on an EBCDIC machine than when run on an ASCII machine. Here is an example adapted from the one in [perlop](#):

```
# EBCDIC-based examples
print "j p \n" ^ " a h"; # prints "JAPH\n"
print "JA" | " ph\n"; # prints "japh\n"
print "JAPH\nJunk" & "\277\277\277\277\277"; # prints "japh\n";
print 'p N$' ^ " E<H\n"; # prints "Perl\n";
```

An interesting property of the 32 C0 control characters in the ASCII table is that they can "literally" be constructed as control characters in perl, e.g. (`chr(0) eq "\c@"`) (`chr(1) eq "\cA"`), and so on.

Perl on EBCDIC machines has been ported to take `"\c@"` to `chr(0)` and `"\cA"` to `chr(1)` as well, but the thirty three characters that result depend on which code page you are using. The table below uses the character names from the previous table but with substitutions such as `s/START OF/S.O./`; `s/END OF/E.O./`; `s/TRANSMISSION/TRANS./`; `s/TABULATION/TAB./`; `s/VERTICAL/VERT./`; `s/HORIZONTAL/HORIZ./`; `s/DEVICE CONTROL/D.C./`; `s/SEPARATOR/SEP./`; `s/NEGATIVE ACKNOWLEDGE/NEG. ACK./`. The POSIX-BC and 1047 sets are identical throughout this range and differ from the 0037 set at only one spot (21 decimal). Note that the LINE FEED character may be generated by `"\cJ"` on ASCII machines but by `"\cU"` on 1047 or POSIX-BC machines and cannot be generated as a `"\c.letter."` control character on 0037 machines. Note also that `"\c\\"` maps to two characters not one.

chr	ord	8859-1	0037	1047 && POSIX-BC

"\c?"	127	<DELETE>	"	" ***><
"\c@"	0	<NULL>	<NULL>	<NULL> ***><
"\cA"	1	<S.O. HEADING>	<S.O. HEADING>	<S.O. HEADING>
"\cB"	2	<S.O. TEXT>	<S.O. TEXT>	<S.O. TEXT>
"\cC"	3	<E.O. TEXT>	<E.O. TEXT>	<E.O. TEXT>
"\cD"	4	<E.O. TRANS.>	<C1 28>	<C1 28>
"\cE"	5	<ENQUIRY>	<HORIZ. TAB.>	<HORIZ. TAB.>
"\cF"	6	<ACKNOWLEDGE>	<C1 6>	<C1 6>
"\cG"	7	<BELL>	<DELETE>	<DELETE>
"\cH"	8	<BACKSPACE>	<C1 23>	<C1 23>
"\cI"	9	<HORIZ. TAB.>	<C1 13>	<C1 13>
"\cJ"	10	<LINE FEED>	<C1 14>	<C1 14>
"\cK"	11	<VERT. TAB.>	<VERT. TAB.>	<VERT. TAB.>
"\cL"	12	<FORM FEED>	<FORM FEED>	<FORM FEED>
"\cM"	13	<CARRIAGE RETURN>	<CARRIAGE RETURN>	<CARRIAGE RETURN>
"\cN"	14	<SHIFT OUT>	<SHIFT OUT>	<SHIFT OUT>
"\cO"	15	<SHIFT IN>	<SHIFT IN>	<SHIFT IN>
"\cP"	16	<DATA LINK ESCAPE>	<DATA LINK ESCAPE>	<DATA LINK ESCAPE>
"\cQ"	17	<D.C. ONE>	<D.C. ONE>	<D.C. ONE>

"\cR"	18	<D.C. TWO>	<D.C. TWO>	<D.C. TWO>	
"\cS"	19	<D.C. THREE>	<D.C. THREE>	<D.C. THREE>	
"\cT"	20	<D.C. FOUR>	<C1 29>	<C1 29>	
"\cU"	21	<NEG. ACK.>	<C1 5>	<LINE FEED>	***
"\cV"	22	<SYNCHRONOUS IDLE>	<BACKSPACE>	<BACKSPACE>	
"\cW"	23	<E.O. TRANS. BLOCK>	<C1 7>	<C1 7>	
"\cX"	24	<CANCEL>	<CANCEL>	<CANCEL>	
"\cY"	25	<E.O. MEDIUM>	<E.O. MEDIUM>	<E.O. MEDIUM>	
"\cZ"	26	<SUBSTITUTE>	<C1 18>	<C1 18>	
"\c["	27	<ESCAPE>	<C1 15>	<C1 15>	
"\c\\"	28	<FILE SEP.>\	<FILE SEP.>\	<FILE SEP.>\	
"\c]"	29	<GROUP SEP.>	<GROUP SEP.>	<GROUP SEP.>	
"\c^"	30	<RECORD SEP.>	<RECORD SEP.>	<RECORD SEP.>	***><
"\c_"	31	<UNIT SEP.>	<UNIT SEP.>	<UNIT SEP.>	***><

FUNCTION DIFFERENCES

`chr()` `chr()` must be given an EBCDIC code number argument to yield a desired character return value on an EBCDIC machine. For example:

```
$CAPITAL_LETTER_A = chr(193);
```

`ord()` `ord()` will return EBCDIC code number values on an EBCDIC machine. For example:

```
$the_number_193 = ord("A");
```

`pack()` The `c` and `C` templates for `pack()` are dependent upon character set encoding. Examples of usage on EBCDIC include:

```
$foo = pack("CCCC",193,194,195,196);
# $foo eq "ABCD"
$foo = pack("C4",193,194,195,196);
# same thing

$foo = pack("ccxccc",193,194,195,196);
# $foo eq "AB\0\0CD"
```

`print()` One must be careful with scalars and strings that are passed to `print` that contain ASCII encodings. One common place for this to occur is in the output of the MIME type header for CGI script writing. For example, many perl programming guides recommend something similar to:

```
print "Content-type:\ttext/html\015\012\015\012";
# this may be wrong on EBCDIC
```

Under the IBM OS/390 USS Web Server or WebSphere on z/OS for example you should instead write that as:

```
print "Content-type:\ttext/html\r\n\r\n"; # OK for DGW et alia
```

That is because the translation from EBCDIC to ASCII is done by the web server in this case (such code will not be appropriate for the Macintosh however). Consult your web server's documentation for further details.

`printf()`

The formats that can convert characters to numbers and vice versa will be different from their ASCII counterparts when executed on an EBCDIC machine. Examples include:

```
printf("%c%c%c",193,194,195); # prints ABC
```

sort() EBCDIC sort results may differ from ASCII sort results especially for mixed case strings. This is discussed in more detail below.

sprintf()

See the discussion of **printf()** above. An example of the use of **sprintf** would be:

```
$CAPITAL_LETTER_A = sprintf("%c",193);
```

unpack()

See the discussion of **pack()** above.

REGULAR EXPRESSION DIFFERENCES

As of perl 5.005_03 the letter range regular expression such as `[A-Z]` and `[a-z]` have been especially coded to not pick up gap characters. For example, characters such as `ô` WITH CIRCUMFLEX that lie between I and J would not be matched by the regular expression range `/[H-K]/`. This works in the other direction, too, if either of the range end points is explicitly numeric: `[\x89-\x91]` will match `\x8e`, even though `\x89` is i and `\x91` is j, and `\x8e` is a gap character from the alphabetic viewpoint.

If you do want to match the alphabet gap characters in a single octet regular expression try matching the hex or octal code such as `/Ë/` on EBCDIC or `/ô/` on ASCII machines to have your regular expression match `ô` WITH CIRCUMFLEX.

Another construct to be wary of is the inappropriate use of hex or octal constants in regular expressions. Consider the following set of subs:

```
sub is_c0 {
    my $char = substr(shift,0,1);
    $char =~ /\000-\037/;
}

sub is_print_ascii {
    my $char = substr(shift,0,1);
    $char =~ /\040-\176/;
}

sub is_delete {
    my $char = substr(shift,0,1);
    $char eq "\177";
}

sub is_c1 {
    my $char = substr(shift,0,1);
    $char =~ /\200-\237/;
}

sub is_latin_1 {
    my $char = substr(shift,0,1);
    $char =~ /\240-\377/;
}
```

The above would be adequate if the concern was only with numeric code points. However, the concern may be with characters rather than code points and on an EBCDIC machine it may be desirable for constructs such as `if (is_print_ascii("A")) {print "A is a printable character\n";}` to print out the expected message. One way to represent the above collection of character classification subs that is capable of working across the four coded character sets discussed in this document is as follows:

```
sub Is_c0 {
    my $char = substr(shift,0,1);
    if (ord('^')==94) { # ascii
        return $char =~ /\000-\037/;
    }
```

```

    }
    if (ord('^')==176) { # 37
        return $char =~ /\[\000-\003\067\055-\057\026\005\045\013-\023\074\075\062
    }
    if (ord('^')==95 || ord('^')==106) { # 1047 || posix-bc
        return $char =~ /\[\000-\003\067\055-\057\026\005\025\013-\023\074\075\062
    }
}

sub Is_print_ascii {
    my $char = substr(shift,0,1);
    $char =~ /[ !"\#$%&'()*+,-.\0-9:;<=>?\@A-Z[\^\_]`a-z{|}~]/;
}

sub Is_delete {
    my $char = substr(shift,0,1);
    if (ord('^')==94) { # ascii
        return $char eq "\177";
    }
    else { # ebcdic
        return $char eq "\007";
    }
}

sub Is_c1 {
    my $char = substr(shift,0,1);
    if (ord('^')==94) { # ascii
        return $char =~ /\[200-\237]/;
    }
    if (ord('^')==176) { # 37
        return $char =~ /\[040-\044\025\006\027\050-\054\011\012\033\060\061\032\
    }
    if (ord('^')==95) { # 1047
        return $char =~ /\[040-\045\006\027\050-\054\011\012\033\060\061\032\063-
    }
    if (ord('^')==106) { # posix-bc
        return $char =~
            /\[040-\045\006\027\050-\054\011\012\033\060\061\032\063-\066\010\070-\
    }
}

sub Is_latin_1 {
    my $char = substr(shift,0,1);
    if (ord('^')==94) { # ascii
        return $char =~ /\[240-\377]/;
    }
    if (ord('^')==176) { # 37
        return $char =~
            /\[101\252\112\261\237\262\152\265\275\264\232\212\137\312\257\274\220\
    }
    if (ord('^')==95) { # 1047
        return $char =~
            /\[101\252\112\261\237\262\152\265\273\264\232\212\260\312\257\274\220\
    }
    if (ord('^')==106) { # posix-bc
        return $char =~

```

```

        / [\101\252\260\261\237\262\320\265\171\264\232\212\272\312\257\241\220\
    }
}

```

Note however that only the `Is_ascii_print()` sub is really independent of coded character set. Another way to write `Is_latin_1()` would be to use the characters in the range explicitly:

```

sub Is_latin_1 {
    my $char = substr(shift,0,1);
    $char =~ /\[ ;ç£¤¥¦§¨ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâ
}

```

Although that form may run into trouble in network transit (due to the presence of 8 bit characters) or on non ISO–Latin character sets.

SOCKETS

Most socket programming assumes ASCII character encodings in network byte order. Exceptions can include CGI script writing under a host web server where the server may take care of translation for you. Most host web servers convert EBCDIC data to ISO–8859–1 or Unicode on output.

SORTING

One big difference between ASCII based character sets and EBCDIC ones are the relative positions of upper and lower case letters and the letters compared to the digits. If sorted on an ASCII based machine the two letter abbreviation for a physician comes before the two letter for drive, that is:

```

@sorted = sort(qw(Dr. dr.)); # @sorted holds ('Dr.', 'dr.') on ASCII,
                             # but ('dr.', 'Dr.') on EBCDIC

```

The property of lower case before uppercase letters in EBCDIC is even carried to the Latin 1 EBCDIC pages such as 0037 and 1047. An example would be that `Ë E WITH DIAERESIS` (203) comes before `ë e WITH DIAERESIS` (235) on an ASCII machine, but the latter (83) comes before the former (115) on an EBCDIC machine. (Astute readers will note that the upper case version of `SMALL LETTER SHARP S` is simply "SS" and that the upper case version of `ÿ Y WITH DIAERESIS` is not in the 0..255 range but it is at `U+x0178` in Unicode, or `"\x{178}"` in a Unicode enabled Perl).

The sort order will cause differences between results obtained on ASCII machines versus EBCDIC machines. What follows are some suggestions on how to deal with these differences.

Ignore ASCII vs. EBCDIC sort differences.

This is the least computationally expensive strategy. It may require some user education.

MONO CASE then sort data.

In order to minimize the expense of mono casing mixed test try to `tr///` towards the character set case most employed within the data. If the data are primarily UPPERCASE non Latin 1 then apply `tr/[a-z]/[A-Z]/` then `sort()`. If the data are primarily lowercase non Latin 1 then apply `tr/[A-Z]/[a-z]/` before sorting. If the data are primarily UPPERCASE and include Latin–1 characters then apply:

```

tr/[a-z]/[A-Z]/;
tr/[ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞ]/[ÄÅÃÄÃÄÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞ]/;
s/ß/SS/g;

```

then `sort()`. Do note however that such Latin–1 manipulation does not address the `ÿ Y WITH DIAERESIS` character that will remain at code point 255 on ASCII machines, but 223 on most EBCDIC machines where it will sort to a place less than the EBCDIC numerals. With a Unicode enabled Perl you might try:

```

tr/^?/\x{178}/;

```

The strategy of mono casing data before sorting does not preserve the case of the data and may not be acceptable for that reason.

Convert, sort data, then re convert.

This is the most expensive proposition that does not employ a network connection.

Perform sorting on one type of machine only.

This strategy can employ a network connection. As such it would be computationally expensive.

TRANSFORMATION FORMATS

There are a variety of ways of transforming data with an intra character set mapping that serve a variety of purposes. Sorting was discussed in the previous section and a few of the other more popular mapping techniques are discussed next.

URL decoding and encoding

Note that some URLs have hexadecimal ASCII code points in them in an attempt to overcome character or protocol limitation issues. For example the tilde character is not on every keyboard hence a URL of the form:

```
http://www.pvhp.com/~pvhp/
```

may also be expressed as either of:

```
http://www.pvhp.com/%7Epvhp/
```

```
http://www.pvhp.com/%7epvhp/
```

where 7E is the hexadecimal ASCII code point for '~'. Here is an example of decoding such a URL under CCSID 1047:

```
$url = 'http://www.pvhp.com/%7Epvhp/';
# this array assumes code page 1047
my @a2e_1047 = (
    0, 1, 2, 3, 55, 45, 46, 47, 22, 5, 21, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 60, 61, 50, 38, 24, 25, 63, 39, 28, 29, 30, 31,
    64, 90, 127, 123, 91, 108, 80, 125, 77, 93, 92, 78, 107, 96, 75, 97,
    240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 122, 94, 76, 126, 110, 111,
    124, 193, 194, 195, 196, 197, 198, 199, 200, 201, 209, 210, 211, 212, 213, 214,
    215, 216, 217, 226, 227, 228, 229, 230, 231, 232, 233, 173, 224, 189, 95, 109,
    121, 129, 130, 131, 132, 133, 134, 135, 136, 137, 145, 146, 147, 148, 149, 150,
    151, 152, 153, 162, 163, 164, 165, 166, 167, 168, 169, 192, 79, 208, 161, 7,
    32, 33, 34, 35, 36, 37, 6, 23, 40, 41, 42, 43, 44, 9, 10, 27,
    48, 49, 26, 51, 52, 53, 54, 8, 56, 57, 58, 59, 4, 20, 62, 255,
    65, 170, 74, 177, 159, 178, 106, 181, 187, 180, 154, 138, 176, 202, 175, 188,
    144, 143, 234, 250, 190, 160, 182, 179, 157, 218, 155, 139, 183, 184, 185, 171,
    100, 101, 98, 102, 99, 103, 158, 104, 116, 113, 114, 115, 120, 117, 118, 119,
    172, 105, 237, 238, 235, 239, 236, 191, 128, 253, 254, 251, 252, 186, 174, 89,
    68, 69, 66, 70, 67, 71, 156, 72, 84, 81, 82, 83, 88, 85, 86, 87,
    140, 73, 205, 206, 203, 207, 204, 225, 112, 221, 222, 219, 220, 141, 142, 223
);
$url =~ s/%([0-9a-fA-F]{2})/pack("c", $a2e_1047[hex($1)])/ge;
```

Conversely, here is a partial solution for the task of encoding such a URL under the 1047 code page:

```
$url = 'http://www.pvhp.com/~pvhp/';
# this array assumes code page 1047
my @e2a_1047 = (
    0, 1, 2, 3, 156, 9, 134, 127, 151, 141, 142, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 157, 10, 8, 135, 24, 25, 146, 143, 28, 29, 30, 31,
    128, 129, 130, 131, 132, 133, 23, 27, 136, 137, 138, 139, 140, 5, 6, 7,
    144, 145, 22, 147, 148, 149, 150, 4, 152, 153, 154, 155, 20, 21, 158, 26,
    32, 160, 226, 228, 224, 225, 227, 229, 231, 241, 162, 46, 60, 40, 43, 124,
```

```

        38,233,234,235,232,237,238,239,236,223, 33, 36, 42, 41, 59, 94,
        45, 47,194,196,192,193,195,197,199,209,166, 44, 37, 95, 62, 63,
        248,201,202,203,200,205,206,207,204, 96, 58, 35, 64, 39, 61, 34,
        216, 97, 98, 99,100,101,102,103,104,105,171,187,240,253,254,177,
        176,106,107,108,109,110,111,112,113,114,170,186,230,184,198,164,
        181,126,115,116,117,118,119,120,121,122,161,191,208, 91,222,174,
        172,163,165,183,169,167,182,188,189,190,221,168,175, 93,180,215,
        123, 65, 66, 67, 68, 69, 70, 71, 72, 73,173,244,246,242,243,245,
        125, 74, 75, 76, 77, 78, 79, 80, 81, 82,185,251,252,249,250,255,
        92,247, 83, 84, 85, 86, 87, 88, 89, 90,178,212,214,210,211,213,
        48, 49, 50, 51, 52, 53, 54, 55, 56, 57,179,219,220,217,218,159
    );
    # The following regular expression does not address the
    # mappings for: ( '.' => '%2E', '/' => '%2F', ':' => '%3A' )
    $url =~ s/([\\t "%&\\(\\),;<=>\\?\\@\\[\\]\\^`{|}~])/sprintf("%02X",$e2a_1047[ord($1)]

```

where a more complete solution would split the URL into components and apply a full `s///` substitution only to the appropriate parts.

In the remaining examples a `@e2a` or `@a2e` array may be employed but the assignment will not be shown explicitly. For code page 1047 you could use the `@a2e_1047` or `@e2a_1047` arrays just shown.

uu encoding and decoding

The `u` template to `pack()` or `unpack()` will render EBCDIC data in EBCDIC characters equivalent to their ASCII counterparts. For example, the following will print "Yes indeed\n" on either an ASCII or EBCDIC computer:

```

    $all_byte_chrs = '';
    for (0..255) { $all_byte_chrs .= chr($_); }
    $uuencode_byte_chrs = pack('u', $all_byte_chrs);
    ($uu = <<'ENDOFHEREDOC') =~ s/^\s*/gm;
    M``$``P0%!@<("0H+#`T.#Q`1$A,4%187&!D:&QP='A\@(2(C)"4F)R@I*BLL
    M+2XO,#$R,S0U-C<X.3H[/#T^/T!!0D-$149'2$E*2TQ-3D]045)35%565UA9
    M6EM<75Y?8&%B8V1E9F=H:6IK;&UN;W!Q<G-T=79W>'EZ>WQ]?G^`@8*#A(6&
    MAXB)BHN,C8Z/D)&2DY25EI>8F9J;G)V>GZ"AHJ.DI::GJ*FJJZRMKJ^PL;*S
    MM+6VM[BYNKN\O;Z_P,'`P\3%QL?(R<K+S,W.S)#1TM/4U=;7V-G:V]S=WM_@
    ?X>+CY.7FY^CIZNOL[>[O\/'R\_3U]O?X^?K[_/W^_P``
    ENDOFHEREDOC
    if ($uuencode_byte_chrs eq $uu) {
        print "Yes ";
    }
    $uudecode_byte_chrs = unpack('u', $uuencode_byte_chrs);
    if ($uudecode_byte_chrs eq $all_byte_chrs) {
        print "indeed\n";
    }

```

Here is a very spartan uudecoder that will work on EBCDIC provided that the `@e2a` array is filled in appropriately:

```

#!/usr/local/bin/perl
@e2a = ( # this must be filled in
);
$_ = <> until ($mode,$file) = /^begin\s*(\d*)\s*(\S*)/;
open(OUT, "> $file") if $file ne "";
while(<>) {
    last if /^end/;
    next if /[a-z]/;

```

```

        next unless int((((($e2a[ord()] - 32) & 077) + 2) / 3) ==
            int(length() / 4);
        print OUT unpack("u", $_);
    }
    close(OUT);
    chmod oct($mode), $file;

```

Quoted-Printable encoding and decoding

On ASCII encoded machines it is possible to strip characters outside of the printable set using:

```

# This QP encoder works on ASCII only
$qp_string =~ s/([=\x00-\x1F\x80-\xFF])/sprintf("=%02X",ord($1))/ge;

```

Whereas a QP encoder that works on both ASCII and EBCDIC machines would look somewhat like the following (where the EBCDIC branch @e2a array is omitted for brevity):

```

if (ord('A') == 65) {      # ASCII
    $delete = "\x7F";      # ASCII
    @e2a = (0 .. 255)      # ASCII to ASCII identity map
}
else {                     # EBCDIC
    $delete = "\x07";      # EBCDIC
    @e2a =                 # EBCDIC to ASCII map (as shown above)
}
$qp_string =~
    s/([^\!\"#$%&'()*+,-./0-9:;<>?\\@A-Z[\\]\\^_`a-z{|}~$delete])/sprintf("=%02X",

```

(although in production code the substitutions might be done in the EBCDIC branch with the @e2a array and separately in the ASCII branch without the expense of the identity map).

Such QP strings can be decoded with:

```

# This QP decoder is limited to ASCII only
$string =~ s/([0-9A-Fa-f][0-9A-Fa-f])/chr hex $1/ge;
$string =~ s/([\\n\\r])+$//;

```

Whereas a QP decoder that works on both ASCII and EBCDIC machines would look somewhat like the following (where the @a2e array is omitted for brevity):

```

$string =~ s/([0-9A-Fa-f][0-9A-Fa-f])/chr $a2e[hex $1]/ge;
$string =~ s/([\\n\\r])+$//;

```

Caesarian ciphers

The practice of shifting an alphabet one or more characters for encipherment dates back thousands of years and was explicitly detailed by Gaius Julius Caesar in his **Gallic Wars** text. A single alphabet shift is sometimes referred to as a rotation and the shift amount is given as a number \$n after the string 'rot' or "rot\$n". Rot0 and rot26 would designate identity maps on the 26 letter English version of the Latin alphabet. Rot13 has the interesting property that alternate subsequent invocations are identity maps (thus rot13 is its own non-trivial inverse in the group of 26 alphabet rotations). Hence the following is a rot13 encoder and decoder that will work on ASCII and EBCDIC machines:

```

#!/usr/local/bin/perl

while(<>){
    tr/n-za-mN-ZA-M/a-zA-Z/;
    print;
}

```

In one-liner form:

```
perl -ne 'tr/n-za-mN-ZA-M/a-zA-Z/;print'
```

Hashing order and checksums

To the extent that it is possible to write code that depends on hashing order there may be differences between hashes as stored on an ASCII based machine and hashes stored on an EBCDIC based machine. XXX

I18N AND L10N

Internationalization(I18N) and localization(L10N) are supported at least in principle even on EBCDIC machines. The details are system dependent and discussed under the [OS ISSUES](#) section below.

MULTI OCTET CHARACTER SETS

Perl may work with an internal UTF–EBCDIC encoding form for wide characters on EBCDIC platforms in a manner analogous to the way that it works with the UTF–8 internal encoding form on ASCII based platforms.

Legacy multi byte EBCDIC code pages XXX.

OS ISSUES

There may be a few system dependent issues of concern to EBCDIC Perl programmers.

OS/400

The PASE environment.

IFS access

XXX.

OS/390, z/OS

Perl runs under Unix Systems Services or USS.

chcp **chcp** is supported as a shell utility for displaying and changing one's code page. See also [chcp](#).

dataset access

For sequential data set access try:

```
my @ds_records = `cat //DSNAME`;
```

or:

```
my @ds_records = `cat //'HLQ.DSNAME'`;
```

See also the OS390::Stdio module on CPAN.

OS/390, z/OS iconv

iconv is supported as both a shell utility and a C RTL routine. See also the iconv(1) and iconv(3) manual pages.

locales On OS/390 or z/OS see [locale](#) for information on locales. The L10N files are in */usr/nls/locale*. `$Config{d_setlocale}` is 'define' on OS/390 or z/OS.

VM/ESA?

XXX.

POSIX–BC?

XXX.

BUGS

This pod document contains literal Latin 1 characters and may encounter translation difficulties. In particular one popular nroff implementation was known to strip accented characters to their unaccented counterparts while attempting to view this document through the **pod2man** program (for example, you may see a plain y rather than one with a diaeresis as in *ÿ*). Another nroff truncated the resultant manpage at the first occurrence of 8 bit characters.

Not all shells will allow multiple `-e` string arguments to perl to be concatenated together properly as recipes 0, 2, 4, 5, and 6 might seem to imply.

SEE ALSO

[*perllocale*](#), [*perlfunc*](#), [*perlunicode*](#), [*utf8*](#).

REFERENCES

<http://anubis.dkuug.dk/i18n/charmaps>

<http://www.unicode.org/>

<http://www.unicode.org/unicode/reports/tr16/>

<http://www.wps.com/texts/codes/> **ASCII: American Standard Code for Information Infiltration** Tom Jennings, September 1999.

The Unicode Standard, Version 3.0 The Unicode Consortium, Lisa Moore ed., ISBN 0-201-61633-5, Addison Wesley Developers Press, February 2000.

CDRA: IBM – Character Data Representation Architecture – Reference and Registry, IBM SC09-2190-00, December 1996.

"Demystifying Character Sets", Andrea Vine, Multilingual Computing & Technology, #26 Vol. 10 Issue 4, August/September 1999; ISSN 1523-0309; Multilingual Computing Inc. Sandpoint ID, USA.

Codes, Ciphers, and Other Cryptic and Clandestine Communication Fred B. Wrixon, ISBN 1-57912-040-7, Black Dog & Leventhal Publishers, 1998.

<http://www.bobbemer.com/P-BIT.HTM> **IBM – EBCDIC and the P-bit; The biggest Computer Goof Ever** Robert Bemer.

HISTORY

15 April 2001: added UTF-8 and UTF-EBCDIC to main table, pvhp.

AUTHOR

Peter Prymmer pvhp@best.com wrote this in 1999 and 2000 with CCSID 0819 and 0037 help from Chris Leach and André Pirard A.Pirard@ulg.ac.be as well as POSIX-BC help from Thomas Dorner Thomas.Dorner@start.de. Thanks also to Vickie Cooper, Philip Newton, William Raffloer, and Joe Smith. Trademarks, registered trademarks, service marks and registered service marks used in this document are the property of their respective owners.

NAME

perlembed – how to embed perl in your C program

DESCRIPTION**PREAMBLE**

Do you want to:

Use C from Perl?

Read [perlxs](#), [perlxs](#), [h2xs](#), [perlguts](#), and [perlapi](#).

Use a Unix program from Perl?

Read about back-quotes and about `system` and `exec` in [perlfunc](#).

Use Perl from Perl?

Read about [do](#) and [eval](#) and [require](#) and [use](#).

Use C from C?

Rethink your design.

Use Perl from C?

Read on...

ROADMAP

- Compiling your C program
- Adding a Perl interpreter to your C program
- Calling a Perl subroutine from your C program
- Evaluating a Perl statement from your C program
- Performing Perl pattern matches and substitutions from your C program
- Fiddling with the Perl stack from your C program
- Maintaining a persistent interpreter
- Maintaining multiple interpreter instances
- Using Perl modules, which themselves use C libraries, from your C program
- Embedding Perl under Win32

Compiling your C program

If you have trouble compiling the scripts in this documentation, you're not alone. The cardinal rule: **COMPILE THE PROGRAMS IN EXACTLY THE SAME WAY THAT YOUR PERL WAS COMPILED.** (Sorry for yelling.)

Also, every C program that uses Perl must link in the *perl library*. What's that, you ask? Perl is itself written in C; the perl library is the collection of compiled C programs that were used to create your perl executable (*/usr/bin/perl* or equivalent). (Corollary: you can't use Perl from your C program unless Perl has been compiled on your machine, or installed properly—that's why you shouldn't blithely copy Perl executables from machine to machine without also copying the *lib* directory.)

When you use Perl from C, your C program will—usually—allocate, "run", and deallocate a *PerlInterpreter* object, which is defined by the perl library.

If your copy of Perl is recent enough to contain this documentation (version 5.002 or later), then the perl library (and *EXTERN.h* and *perl.h*, which you'll also need) will reside in a directory that looks like this:

```
/usr/local/lib/perl5/your_architecture_here/CORE
```

or perhaps just

```
/usr/local/lib/perl5/CORE
```

or maybe something like

```
/usr/opt/perl5/CORE
```

Execute this statement for a hint about where to find CORE:

```
perl -MConfig -e 'print $Config{archlib}'
```

Here's how you'd compile the example in the next section, [Adding a Perl interpreter to your C program](#), on my Linux box:

```
% gcc -O2 -Dbool=char -DHAS_BOOL -I/usr/local/include
-I/usr/local/lib/perl5/i586-linux/5.003/CORE
-L/usr/local/lib/perl5/i586-linux/5.003/CORE
-o interp interp.c -lperl -lm
```

(That's all one line.) On my DEC Alpha running old 5.003_05, the incantation is a bit different:

```
% cc -O2 -Olimit 2900 -DSTANDARD_C -I/usr/local/include
-I/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE
-L/usr/local/lib/perl5/alpha-dec_osf/5.00305/CORE -L/usr/local/lib
-D__LANGUAGE_C__ -D_NO_PROTO -o interp interp.c -lperl -lm
```

How can you figure out what to add? Assuming your Perl is post-5.001, execute a `perl -V` command and pay special attention to the "cc" and "ccflags" information.

You'll have to choose the appropriate compiler (*cc*, *gcc*, et al.) for your machine: `perl -MConfig -e 'print $Config{cc}'` will tell you what to use.

You'll also have to choose the appropriate library directory (*/usr/local/lib/...*) for your machine. If your compiler complains that certain functions are undefined, or that it can't locate *-lperl*, then you need to change the path following the *-L*. If it complains that it can't find *EXTERN.h* and *perl.h*, you need to change the path following the *-I*.

You may have to add extra libraries as well. Which ones? Perhaps those printed by

```
perl -MConfig -e 'print $Config{libs}'
```

Provided your perl binary was properly configured and installed the **ExtUtils::Embed** module will determine all of this information for you:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

If the **ExtUtils::Embed** module isn't part of your Perl distribution, you can retrieve it from <http://www.perl.com/perl/CPAN/modules/by-module/ExtUtils/> (If this documentation came from your Perl distribution, then you're running 5.004 or better and you already have it.)

The **ExtUtils::Embed** kit on CPAN also contains all source code for the examples in this document, tests, additional examples and other information you may find useful.

Adding a Perl interpreter to your C program

In a sense, perl (the C program) is a good example of embedding Perl (the language), so I'll demonstrate embedding with *miniperlmain.c*, included in the source distribution. Here's a bastardized, nonportable version of *miniperlmain.c* containing the essentials of embedding:

```
#include <EXTERN.h> /* from the Perl distribution */
#include <perl.h> /* from the Perl distribution */

static PerlInterpreter *my_perl; /* The Perl interpreter */

int main(int argc, char **argv, char **env)
```

```

{
    my_perl = perl_alloc();
    perl_construct(my_perl);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    perl_parse(my_perl, NULL, argc, argv, (char **)NULL);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);
}

```

Notice that we don't use the `env` pointer. Normally handed to `perl_parse` as its final argument, `env` here is replaced by `NULL`, which means that the current environment will be used.

Now compile this program (I'll call it *interp.c*) into an executable:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

After a successful compilation, you'll be able to use *interp* just like *perl* itself:

```

% interp
print "Pretty Good Perl \n";
print "10890 - 9801 is ", 10890 - 9801;
<CTRL-D>
Pretty Good Perl
10890 - 9801 is 1089

```

or

```

% interp -e 'printf("%x", 3735928559)'
deadbeef

```

You can also read and execute Perl statements from a file while in the midst of your C program, by placing the filename in `argv[1]` before calling *perl_run*.

Calling a Perl subroutine from your C program

To call individual Perl subroutines, you can use any of the **call_*** functions documented in [perlcalls](#). In this example we'll use `call_argv`.

That's shown below, in a program I'll call *showtime.c*.

```

#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

int main(int argc, char **argv, char **env)
{
    char *args[] = { NULL };
    my_perl = perl_alloc();
    perl_construct(my_perl);

    perl_parse(my_perl, NULL, argc, argv, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

    /** skipping perl_run() ***/

    call_argv("showtime", G_DISCARD | G_NOARGS, args);

    perl_destruct(my_perl);
    perl_free(my_perl);
}

```

where *showtime* is a Perl subroutine that takes no arguments (that's the *G_NOARGS*) and for which I'll

ignore the return value (that's the *G_DISCARD*). Those flags, and others, are discussed in [perlcall](#).

I'll define the *showtime* subroutine in a file called *showtime.pl*:

```
print "I shan't be printed.";

sub showtime {
    print time;
}
```

Simple enough. Now compile and run:

```
% cc -o showtime showtime.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
% showtime showtime.pl
818284590
```

yielding the number of seconds that elapsed between January 1, 1970 (the beginning of the Unix epoch), and the moment I began writing this sentence.

In this particular case we don't have to call *perl_run*, as we set the *PL_exit_flag* *PERL_EXIT_DESTRUCT_END* which executes *END* blocks in *perl_destruct*.

If you want to pass arguments to the Perl subroutine, you can add strings to the *NULL*-terminated *args* list passed to *call_argv*. For other data types, or to examine return values, you'll need to manipulate the Perl stack. That's demonstrated in [Fiddling with the Perl stack from your C program](#).

Evaluating a Perl statement from your C program

Perl provides two API functions to evaluate pieces of Perl code. These are [eval_sv](#) and [eval_pv](#).

Arguably, these are the only routines you'll ever need to execute snippets of Perl code from within your C program. Your code can be as long as you wish; it can contain multiple statements; it can employ [use](#), [require](#), and [do](#) to include external Perl files.

eval_pv lets us evaluate individual Perl strings, and then extract variables for coercion into C types. The following program, *string.c*, executes three Perl strings, extracting an *int* from the first, a *float* from the second, and a *char ** from the third.

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

main (int argc, char **argv, char **env)
{
    STRLEN n_a;
    char *embedding[] = { "", "-e", "0" };

    my_perl = perl_alloc();
    perl_construct( my_perl );

    perl_parse(my_perl, NULL, 3, embedding, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    perl_run(my_perl);

    /** Treat $a as an integer **/
    eval_pv("$a = 3; $a **= 2", TRUE);
    printf("a = %d\n", SvIV(get_sv("a", FALSE)));

    /** Treat $a as a float **/
    eval_pv("$a = 3.14; $a **= 2", TRUE);
    printf("a = %f\n", SvNV(get_sv("a", FALSE)));

    /** Treat $a as a string **/
```

```

    eval_pv("$a = 'rekcaH lreP rehtonA tsuJ'; $a = reverse($a);", TRUE);
    printf("a = %s\n", SvPV(get_sv("a", FALSE), n_a));

    perl_destruct(my_perl);
    perl_free(my_perl);
}

```

All of those strange functions with *sv* in their names help convert Perl scalars to C types. They're described in [perlguts](#) and [perlapi](#).

If you compile and run *string.c*, you'll see the results of using *SvIV()* to create an int, *SvNV()* to create a float, and *SvPV()* to create a string:

```

a = 9
a = 9.859600
a = Just Another Perl Hacker

```

In the example above, we've created a global variable to temporarily store the computed value of our eval'd expression. It is also possible and in most cases a better strategy to fetch the return value from *eval_pv()* instead. Example:

```

...
STRLEN n_a;
SV *val = eval_pv("reverse 'rekcaH lreP rehtonA tsuJ'", TRUE);
printf("%s\n", SvPV(val, n_a));
...

```

This way, we avoid namespace pollution by not creating global variables and we've simplified our code as well.

Performing Perl pattern matches and substitutions from your C program

The *eval_sv()* function lets us evaluate strings of Perl code, so we can define some functions that use it to "specialize" in matches and substitutions: *match()*, *substitute()*, and *matches()*.

```
I32 match(SV *string, char *pattern);
```

Given a string and a pattern (e.g., *m/clasp/* or */\b\w*\b/*, which in your C program might appear as *"^\\b\\w*\\b/"*), *match()* returns 1 if the string matches the pattern and 0 otherwise.

```
int substitute(SV **string, char *pattern);
```

Given a pointer to an SV and an *=~* operation (e.g., *s/bob/robert/g* or *tr[A-Z][a-z]*), *substitute()* modifies the string within the AV at according to the operation, returning the number of substitutions made.

```
int matches(SV *string, char *pattern, AV **matches);
```

Given an SV, a pattern, and a pointer to an empty AV, *matches()* evaluates *\$string =~ \$pattern* in a list context, and fills in *matches* with the array elements, returning the number of matches found.

Here's a sample program, *match.c*, that uses all three (long lines have been wrapped here):

```

#include <EXTERN.h>
#include <perl.h>

/** my_eval_sv(code, error_check)
** kinda like eval_sv(),
** but we pop the return value off the stack
**/
SV* my_eval_sv(SV *sv, I32 croak_on_error)
{
    dSP;
    SV* retval;

```

```

    STRLEN n_a;

    PUSHMARK(SP);
    eval_sv(sv, G_SCALAR);

    SPAGAIN;
    retval = POPs;
    PUTBACK;

    if (croak_on_error && SvTRUE(ERRSV))
        croak(SvPVx(ERRSV, n_a));

    return retval;
}

/** match(string, pattern)
**
** Used for matches in a scalar context.
**
** Returns 1 if the match was successful; 0 otherwise.
**/

I32 match(SV *string, char *pattern)
{
    SV *command = NEWSV(1099, 0), *retval;
    STRLEN n_a;

    sv_setpvf(command, "my $string = '%s'; $string =~ %s",
               SvPV(string, n_a), pattern);

    retval = my_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    return SvIV(retval);
}

/** substitute(string, pattern)
**
** Used for =~ operations that modify their left-hand side (s/// and tr///)
**
** Returns the number of successful matches, and
** modifies the input string if there were any.
**/

I32 substitute(SV **string, char *pattern)
{
    SV *command = NEWSV(1099, 0), *retval;
    STRLEN n_a;

    sv_setpvf(command, "$string = '%s'; ($string =~ %s)",
               SvPV(*string, n_a), pattern);

    retval = my_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    *string = get_sv("string", FALSE);
    return SvIV(retval);
}

/** matches(string, pattern, matches)
**
** Used for matches in a list context.

```

```

**
** Returns the number of matches,
** and fills in **matches with the matching substrings
**/

I32 matches(SV *string, char *pattern, AV **match_list)
{
    SV *command = NEWSV(1099, 0);
    I32 num_matches;
    STRLEN n_a;

    sv_setpvf(command, "my $string = '%s'; @array = ($string =~ %s)",
                SvPV(string, n_a), pattern);

    my_eval_sv(command, TRUE);
    SvREFCNT_dec(command);

    *match_list = get_av("array", FALSE);
    num_matches = av_len(*match_list) + 1; /** assume $[ is 0 **/
    return num_matches;
}

main (int argc, char **argv, char **env)
{
    PerlInterpreter *my_perl = perl_alloc();
    char *embedding[] = { "", "-e", "0" };
    AV *match_list;
    I32 num_matches, i;
    SV *text = NEWSV(1099, 0);
    STRLEN n_a;

    perl_construct(my_perl);
    perl_parse(my_perl, NULL, 3, embedding, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;

    sv_setpv(text, "When he is at a convenience store and the bill comes to some amo

    if (match(text, "m/quarter/")) /** Does text contain 'quarter'? **/
        printf("match: Text contains the word 'quarter'.\n\n");
    else
        printf("match: Text doesn't contain the word 'quarter'.\n\n");
    if (match(text, "m/eighth/")) /** Does text contain 'eighth'? **/
        printf("match: Text contains the word 'eighth'.\n\n");
    else
        printf("match: Text doesn't contain the word 'eighth'.\n\n");

    /** Match all occurrences of /wi../ **/
    num_matches = matches(text, "m/(wi..)/g", &match_list);
    printf("matches: m/(wi..)/g found %d matches...\n", num_matches);

    for (i = 0; i < num_matches; i++)
        printf("match: %s\n", SvPV(*av_fetch(match_list, i, FALSE), n_a));
    printf("\n");

    /** Remove all vowels from text **/
    num_matches = substitute(&text, "s/[aeiou]//gi");
    if (num_matches) {
        printf("substitute: s/[aeiou]//gi...%d substitutions made.\n",
                num_matches);
    }
}

```



```

        printf("Now text is: %s\n\n", SvPV(text,n_a));
    }

    /** Attempt a substitution **/
    if (!substitute(&text, "s/Perl/C/")) {
        printf("substitute: s/Perl/C...No substitution made.\n\n");
    }

    SvREFCNT_dec(text);
    PL_perl_destruct_level = 1;
    perl_destruct(my_perl);
    perl_free(my_perl);
}

```

which produces the output (again, long lines have been wrapped here)

```

match: Text contains the word 'quarter'.

match: Text doesn't contain the word 'eighth'.

matches: m/(wi..)/g found 2 matches...
match: will
match: with

substitute: s/[aeiou]/gi...139 substitutions made.
Now text is: Whn h s t  cnvnnc str nd th bll cms t sm mnt lk 76 cnts,
Mynrd s wr tht thr s smthng h *shld* d, smthng tht wll nbl hm t gt bck
qrtr, bt h hs n d *wht*.  H fmbls thrgh hs rd sqzy chngprs nd gvs th by
thr xtr pnns wth hs dllr, hpng tht h mght lck nt th crct mnt.  Th by gvs
hm bck tw f hs wn pnns nd thn th bg shny qrtr tht s hs prz. -RCHH

substitute: s/Perl/C...No substitution made.

```

Fiddling with the Perl stack from your C program

When trying to explain stacks, most computer science textbooks mumble something about spring-loaded columns of cafeteria plates: the last thing you pushed on the stack is the first thing you pop off. That'll do for our purposes: your C program will push some arguments onto "the Perl stack", shut its eyes while some magic happens, and then pop the results—the return value of your Perl subroutine—off the stack.

First you'll need to know how to convert between C types and Perl types, with `newSViv()` and `sv_setnv()` and `newAV()` and all their friends. They're described in [perlguts](#) and [perlapi](#).

Then you'll need to know how to manipulate the Perl stack. That's described in [perlcall](#).

Once you've understood those, embedding Perl in C is easy.

Because C has no builtin function for integer exponentiation, let's make Perl's `**` operator available to it (this is less useful than it sounds, because Perl implements `**` with C's `pow()` function). First I'll create a stub exponentiation function in *power.pl*:

```

sub expo {
    my ($a, $b) = @_;
    return $a ** $b;
}

```

Now I'll create a C program, *power.c*, with a function `PerlPower()` that contains all the perlguits necessary to push the two arguments into `expo()` and to pop the return value out. Take a deep breath...

```

#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;

```

```

static void
PerlPower(int a, int b)
{
    dSP;                                /* initialize stack pointer */
    ENTER;                              /* everything created after here */
    SAVETMPS;                           /* ...is a temporary variable. */
    PUSHMARK(SP);                       /* remember the stack pointer */
    XPUSHs(sv_2mortal(newSViv(a))); /* push the base onto the stack */
    XPUSHs(sv_2mortal(newSViv(b))); /* push the exponent onto stack */
    PUTBACK;                            /* make local stack pointer global */
    call_pv("expo", G_SCALAR);          /* call the function */
    SPAGAIN;                            /* refresh stack pointer */
                                        /* pop the return value from stack */
    printf ("%d to the %dth power is %d.\n", a, b, POPI);
    PUTBACK;
    FREETMPS;                           /* free that return value */
    LEAVE;                              /* ...and the XPUSHed "mortal" args.*/
}

int main (int argc, char **argv, char **env)
{
    char *my_argv[] = { "", "power.pl" };

    my_perl = perl_alloc();
    perl_construct( my_perl );

    perl_parse(my_perl, NULL, 2, my_argv, (char **)NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    perl_run(my_perl);

    PerlPower(3, 4);                    /**** Compute 3 ** 4 ****/

    perl_destruct(my_perl);
    perl_free(my_perl);
}

```

Compile and run:

```

% cc -o power power.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
% power
3 to the 4th power is 81.

```

Maintaining a persistent interpreter

When developing interactive and/or potentially long-running applications, it's a good idea to maintain a persistent interpreter rather than allocating and constructing a new interpreter multiple times. The major reason is speed: since Perl will only be loaded into memory once.

However, you have to be more cautious with namespace and variable scoping when using a persistent interpreter. In previous examples we've been using global variables in the default package `main`. We knew exactly what code would be run, and assumed we could avoid variable collisions and outrageous symbol table growth.

Let's say your application is a server that will occasionally run Perl code from some arbitrary file. Your server has no way of knowing what code it's going to run. Very dangerous.

If the file is pulled in by `perl_parse()`, compiled into a newly constructed interpreter, and subsequently cleaned out with `perl_destruct()` afterwards, you're shielded from most namespace troubles.

One way to avoid namespace collisions in this scenario is to translate the filename into a guaranteed-unique package name, and then compile the code into that package using *eval*. In the example below, each file will

only be compiled once. Or, the application might choose to clean out the symbol table associated with the file after it's no longer needed. Using *call_argv*, We'll call the subroutine `Embed::Persistent::eval_file` which lives in the file `persistent.pl` and pass the filename and boolean cleanup/cache flag as arguments.

Note that the process will continue to grow for each file that it uses. In addition, there might be AUTOLOADED subroutines and other conditions that cause Perl's symbol table to grow. You might want to add some logic that keeps track of the process size, or restarts itself after a certain number of requests, to ensure that memory consumption is minimized. You'll also want to scope your variables with *my* whenever possible.

```
package Embed::Persistent;
#persistent.pl

use strict;
our %Cache;
use Symbol qw(delete_package);

sub valid_package_name {
    my($string) = @_ ;
    $string =~ s/([^\A-Za-z0-9\_\-])/sprintf("_%2x",unpack("C",$1))/eg;
    # second pass only for words starting with a digit
    $string =~ s|/(\d)|sprintf("/_%2x",unpack("C",$1))|eg;

    # Dress it up as a real package name
    $string =~ s|/|::|g;
    return "Embed" . $string;
}

sub eval_file {
    my($filename, $delete) = @_ ;
    my $package = valid_package_name($filename);
    my $mtime = -M $filename;
    if(defined $Cache{$package}{mtime}
        &&
        $Cache{$package}{mtime} <= $mtime)
    {
        # we have compiled this subroutine already,
        # it has not been updated on disk, nothing left to do
        print STDERR "already compiled $package->handler\n";
    }
    else {
        local *FH;
        open FH, $filename or die "open '$filename' $!";
        local($/) = undef;
        my $sub = <FH>;
        close FH;

        #wrap the code into a subroutine inside our unique package
        my $eval = qq{package $package; sub handler { $sub; }};
        {
            # hide our variables within this block
            my($filename,$mtime,$package,$sub);
            eval $eval;
        }
        die $@ if $@;

        #cache it unless we're cleaning out each time
    }
}
```

```

    $Cache{$package}{mtime} = $mtime unless $delete;
}

eval {$package->handler;};
die $@ if $@;

delete_package($package) if $delete;

#take a look if you want
#print Devel::Symdump->rnew($package)->as_string, $/;
}

1;

__END__

/* persistent.c */
#include <EXTERN.h>
#include <perl.h>

/* 1 = clean out filename's symbol table after each request, 0 = don't */
#ifndef DO_CLEAN
#define DO_CLEAN 0
#endif

static PerlInterpreter *perl = NULL;

int
main(int argc, char **argv, char **env)
{
    char *embedding[] = { "", "persistent.pl" };
    char *args[] = { "", DO_CLEAN, NULL };
    char filename [1024];
    int exitstatus = 0;
    STRLEN n_a;

    if((perl = perl_alloc()) == NULL) {
        fprintf(stderr, "no memory!");
        exit(1);
    }
    perl_construct(perl);

    exitstatus = perl_parse(perl, NULL, 2, embedding, NULL);
    PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
    if(!exitstatus) {
        exitstatus = perl_run(perl);

        while(sprintf("Enter file name: ") && gets(filename)) {
            /* call the subroutine, passing it the filename as an argument */
            args[0] = filename;
            call_argv("Embed::Persistent::eval_file",
                     G_DISCARD | G_EVAL, args);

            /* check $@ */
            if(SvTRUE(ERRSV))
                fprintf(stderr, "eval error: %s\n", SvPV(ERRSV,n_a));
        }
    }

    PL_perl_destruct_level = 0;
    perl_destruct(perl);

```

```
    perl_free(perl);
    exit(exitstatus);
}
```

Now compile:

```
% cc -o persistent persistent.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Here's an example script file:

```
#test.pl
my $string = "hello";
foo($string);

sub foo {
    print "foo says: @_\\n";
}
```

Now run:

```
% persistent
Enter file name: test.pl
foo says: hello
Enter file name: test.pl
already compiled Embed::test_2epl->handler
foo says: hello
Enter file name: ^C
```

Execution of END blocks

Traditionally END blocks have been executed at the end of the perl_run. This causes problems for applications that never call perl_run. Since perl 5.7.2 you can specify `PL_exit_flags |= PERL_EXIT_DESTRUCT_END` to get the new behaviour. This also enables the running of END blocks if the perl_prase fails and perl_destruct will return the exit value.

Maintaining multiple interpreter instances

Some rare applications will need to create more than one interpreter during a session. Such an application might sporadically decide to release any resources associated with the interpreter.

The program must take care to ensure that this takes place *before* the next interpreter is constructed. By default, when perl is not built with any special options, the global variable `PL_perl_destruct_level` is set to 0, since extra cleaning isn't usually needed when a program only ever creates a single interpreter in its entire lifetime.

Setting `PL_perl_destruct_level` to 1 makes everything squeaky clean:

```
PL_perl_destruct_level = 1;

while(1) {
    ...
    /* reset global variables here with PL_perl_destruct_level = 1 */
    perl_construct(my_perl);
    ...
    /* clean and reset everything during perl_destruct */
    perl_destruct(my_perl);
    perl_free(my_perl);
    ...
    /* let's go do it again! */
}
```

When `perl_destruct()` is called, the interpreter's syntax parse tree and symbol tables are cleaned up, and global variables are reset.

Now suppose we have more than one interpreter instance running at the same time. This is feasible, but only if you used the Configure option `-Dusemultiplicity` or the options `-Dusethreads` `-Duseithreads` when building Perl. By default, enabling one of these Configure options sets the per-interpreter global variable `PL_perl_destruct_level` to 1, so that thorough cleaning is automatic.

Using `-Dusethreads` `-Duseithreads` rather than `-Dusemultiplicity` is more appropriate if you intend to run multiple interpreters concurrently in different threads, because it enables support for linking in the thread libraries of your system with the interpreter.

Let's give it a try:

```
#include <EXTERN.h>
#include <perl.h>

/* we're going to embed two interpreters */
/* we're going to embed two interpreters */

#define SAY_HELLO "-e", "print qq(Hi, I'm $^X\n)"

int main(int argc, char **argv, char **env)
{
    PerlInterpreter
        *one_perl = perl_alloc(),
        *two_perl = perl_alloc();
    char *one_args[] = { "one_perl", SAY_HELLO };
    char *two_args[] = { "two_perl", SAY_HELLO };

    PERL_SET_CONTEXT(one_perl);
    perl_construct(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_construct(two_perl);

    PERL_SET_CONTEXT(one_perl);
    perl_parse(one_perl, NULL, 3, one_args, (char **)NULL);
    PERL_SET_CONTEXT(two_perl);
    perl_parse(two_perl, NULL, 3, two_args, (char **)NULL);

    PERL_SET_CONTEXT(one_perl);
    perl_run(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_run(two_perl);

    PERL_SET_CONTEXT(one_perl);
    perl_destruct(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_destruct(two_perl);

    PERL_SET_CONTEXT(one_perl);
    perl_free(one_perl);
    PERL_SET_CONTEXT(two_perl);
    perl_free(two_perl);
}
```

Note the calls to `PERL_SET_CONTEXT()`. These are necessary to initialize the global state that tracks which interpreter is the "current" one on the particular process or thread that may be running it. It should always be used if you have more than one interpreter and are making perl API calls on both interpreters in an interleaved fashion.

`PERL_SET_CONTEXT(interp)` should also be called whenever `interp` is used by a thread that did not create it (using either `perl_alloc()`, or the more esoteric `perl_clone()`).

Compile as usual:

```
% cc -o multiplicity multiplicity.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Run it, Run it:

```
% multiplicity
Hi, I'm one_perl
Hi, I'm two_perl
```

Using Perl modules, which themselves use C libraries, from your C program

If you've played with the examples above and tried to embed a script that *use()*s a Perl module (such as *Socket*) which itself uses a C or C++ library, this probably happened:

```
Can't load module Socket, dynamic loading not available in this perl.
(You may need to build a new perl executable which either supports
dynamic loading or has the Socket module statically linked into it.)
```

What's wrong?

Your interpreter doesn't know how to communicate with these extensions on its own. A little glue will help. Up until now you've been calling *perl_parse()*, handing it NULL for the second argument:

```
perl_parse(my_perl, NULL, argc, my_argv, NULL);
```

That's where the glue code can be inserted to create the initial contact between Perl and linked C/C++ routines. Let's take a look some pieces of *perlmain.c* to see how Perl does this:

```
static void xs_init (pTHX);

EXTERN_C void boot_DynaLoader (pTHX_ CV* cv);
EXTERN_C void boot_Socket (pTHX_ CV* cv);

EXTERN_C void
xs_init(pTHX)
{
    char *file = __FILE__;
    /* DynaLoader is a special case */
    newXS("DynaLoader::boot_DynaLoader", boot_DynaLoader, file);
    newXS("Socket::bootstrap", boot_Socket, file);
}
```

Simply put: for each extension linked with your Perl executable (determined during its initial configuration on your computer or when adding a new extension), a Perl subroutine is created to incorporate the extension's routines. Normally, that subroutine is named *Module::bootstrap()* and is invoked when you say *use Module*. In turn, this hooks into an XSUB, *boot_Module*, which creates a Perl counterpart for each of the extension's XSUBs. Don't worry about this part; leave that to the *xsubpp* and extension authors. If your extension is dynamically loaded, DynaLoader creates *Module::bootstrap()* for you on the fly. In fact, if you have a working DynaLoader then there is rarely any need to link in any other extensions statically.

Once you have this code, slap it into the second argument of *perl_parse()*:

```
perl_parse(my_perl, xs_init, argc, my_argv, NULL);
```

Then compile:

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`

% interp
use Socket;
use SomeDynamicallyLoadedModule;
```

```
print "Now I can use extensions!\n"
```

ExtUtils::Embed can also automate writing the *xs_init* glue code.

```
% perl -MExtUtils::Embed -e xsinit -- -o perlxs1.c
% cc -c perlxs1.c `perl -MExtUtils::Embed -e ccopts`
% cc -c interp.c `perl -MExtUtils::Embed -e ccopts`
% cc -o interp perlxs1.o interp.o `perl -MExtUtils::Embed -e ldopts`
```

Consult [perlxs](#), [perlguts](#), and [perlapi](#) for more details.

Embedding Perl under Win32

In general, all of the source code shown here should work unmodified under Windows.

However, there are some caveats about the command-line examples shown. For starters, backticks won't work under the Win32 native command shell. The ExtUtils::Embed kit on CPAN ships with a script called **genmake**, which generates a simple makefile to build a program from a single C source file. It can be used like this:

```
C:\ExtUtils-Embed\eg> perl genmake interp.c
C:\ExtUtils-Embed\eg> nmake
C:\ExtUtils-Embed\eg> interp -e "print qq{I'm embedded in Win32!\n}"
```

You may wish to use a more robust environment such as the Microsoft Developer Studio. In this case, run this to generate perlxs1.c:

```
perl -MExtUtils::Embed -e xsinit
```

Create a new project and Insert – Files into Project: perlxs1.c, perl.lib, and your own source files, e.g. interp.c. Typically you'll find perl.lib in **C:\perl\lib\CORE**, if not, you should see the **CORE** directory relative to perl -V:archlib. The studio will also need this path so it knows where to find Perl include files. This path can be added via the Tools – Options – Directories menu. Finally, select Build – Build interp.exe and you're ready to go.

MORAL

You can sometimes *write faster code* in C, but you can always *write code faster* in Perl. Because you can use each from the other, combine them as you wish.

AUTHOR

Jon Orwant <orwant@tpj.com> and Doug MacEachern <doug@osf.org>, with small contributions from Tim Bunce, Tom Christiansen, Guy Decoux, Hallvard Furuseth, Dov Grobgeld, and Ilya Zakharevich.

Doug MacEachern has an article on embedding in Volume 1, Issue 4 of The Perl Journal (<http://www.tpj.com/>). Doug is also the developer of the most widely-used Perl embedding: the mod_perl system (perl.apache.org), which embeds Perl in the Apache web server. Oracle, Binary Evolution, ActiveState, and Ben Sugars's nsapi_perl have used this model for Oracle, Netscape and Internet Information Server Perl plugins.

July 22, 1998

COPYRIGHT

Copyright (C) 1995, 1996, 1997, 1998 Doug MacEachern and Jon Orwant. All Rights Reserved.

Permission is granted to make and distribute verbatim copies of this documentation provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that they are marked clearly as modified versions, that the authors' names and title are unchanged (though subtitles and additional authors' names may be added), and that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under

the above conditions for modified versions.

NAME

perlfaq – frequently asked questions about Perl (\$Date: 2002/03/11 21:32:23 \$)

DESCRIPTION

The perlfaq is structured into the following documents:

perlfaq: Structural overview of the FAQ.

This document.

perlfaq1: General Questions About Perl

Very general, high-level questions about Perl.

- What is Perl?
- Who supports Perl? Who develops it? Why is it free?
- Which version of Perl should I use?
- What are perl4 and perl5?
- What is perl6?
- How stable is Perl?
- Is Perl difficult to learn?
- How does Perl compare with other languages like Java, Python, REXX, Scheme, or Tcl?
- Can I do [task] in Perl?
- When shouldn't I program in Perl?
- What's the difference between "perl" and "Perl"?
- Is it a Perl program or a Perl script?
- What is a JAPH?
- Where can I get a list of Larry Wall witticisms?
- How can I convince my sysadmin/supervisor/employees to use version 5/5.6.1/Perl instead of some other language?

perlfaq2: Obtaining and Learning about Perl

Where to find source and documentation for Perl, support, and related matters.

- What machines support Perl? Where do I get it?
- How can I get a binary version of Perl?
- I don't have a C compiler on my system. How can I compile perl?
- I copied the Perl binary from one machine to another, but scripts don't work.
- I grabbed the sources and tried to compile but gdbm/dynamic loading/malloc/linking/... failed. How do I make it work?
- What modules and extensions are available for Perl? What is CPAN? What does CPAN/src/... mean?
- Is there an ISO or ANSI certified version of Perl?
- Where can I get information on Perl?
- What are the Perl newsgroups on Usenet? Where do I post questions?

- Where should I post source code?
- Perl Books
- Perl in Magazines
- Perl on the Net: FTP and WWW Access
- What mailing lists are there for Perl?
- Archives of comp.lang.perl.misc
- Where can I buy a commercial version of Perl?
- Where do I send bug reports?
- What is perl.com? Perl Mongers? pm.org? perl.org? cpan.org?

perlfaq3: Programming Tools

Programmer tools and programming support.

- How do I do (anything)?
- How can I use Perl interactively?
- Is there a Perl shell?
- How do I debug my Perl programs?
- How do I profile my Perl programs?
- How do I cross-reference my Perl programs?
- Is there a pretty-printer (formatter) for Perl?
- Is there a ctags for Perl?
- Is there an IDE or Windows Perl Editor?
- Where can I get Perl macros for vi?
- Where can I get perl-mode for emacs?
- How can I use curses with Perl?
- How can I use X or Tk with Perl?
- How can I generate simple menus without using CGI or Tk?
- How can I make my Perl program run faster?
- How can I make my Perl program take less memory?
- Is it unsafe to return a pointer to local data?
- How can I free an array or hash so my program shrinks?
- How can I make my CGI script more efficient?
- How can I hide the source for my Perl program?
- How can I compile my Perl program into byte code or C?
- How can I compile Perl into Java?
- How can I get `#!perl` to work on [MS-DOS,NT,...]?
- Can I write useful Perl programs on the command line?

- Why don't Perl one-liners work on my DOS/Mac/VMS system?
- Where can I learn about CGI or Web programming in Perl?
- Where can I learn about object-oriented Perl programming?
- Where can I learn about linking C with Perl? [h2xs, xsubpp]
- I've read perlembed, perl guts, etc., but I can't embed perl in my C program; what am I doing wrong?
- When I tried to run my script, I got this message. What does it mean?
- What's MakeMaker?

perlfaq4: Data Manipulation

Manipulating numbers, dates, strings, arrays, hashes, and miscellaneous data issues.

- Why am I getting long decimals (eg, 19.9499999999999) instead of the numbers I should be getting (eg, 19.95)?
- Why isn't my octal data interpreted correctly?
- Does Perl have a `round()` function? What about `ceil()` and `floor()`? Trig functions?
- How do I convert between numeric representations?
- Why doesn't & work the way I want it to?
- How do I multiply matrices?
- How do I perform an operation on a series of integers?
- How can I output Roman numerals?
- Why aren't my random numbers random?
- How do I get a random number between X and Y?
- How do I find the week-of-the-year/day-of-the-year?
- How do I find the current century or millennium?
- How can I compare two dates and find the difference?
- How can I take a string and turn it into epoch seconds?
- How can I find the Julian Day?
- How do I find yesterday's date?
- Does Perl have a Year 2000 problem? Is Perl Y2K compliant?
- How do I validate input?
- How do I unescape a string?
- How do I remove consecutive pairs of characters?
- How do I expand function calls in a string?
- How do I find matching/nesting anything?
- How do I reverse a string?
- How do I expand tabs in a string?
- How do I reformat a paragraph?

- How can I access/change the first N letters of a string?
- How do I change the Nth occurrence of something?
- How can I count the number of occurrences of a substring within a string?
- How do I capitalize all the words on one line?
- How can I split a [character] delimited string except when inside [character]? (Comma-separated files)
- How do I strip blank space from the beginning/end of a string?
- How do I pad a string with blanks or pad a number with zeroes?
- How do I extract selected columns from a string?
- How do I find the soundex value of a string?
- How can I expand variables in text strings?
- What's wrong with always quoting "\$vars"?
- Why don't my <<HERE documents work?
- What is the difference between a list and an array?
- What is the difference between `$array[1]` and `@array[1]`?
- How can I remove duplicate elements from a list or array?
- How can I tell whether a certain element is contained in a list or array?
- How do I compute the difference of two arrays? How do I compute the intersection of two arrays?
- How do I test whether two arrays or hashes are equal?
- How do I find the first array element for which a condition is true?
- How do I handle linked lists?
- How do I handle circular lists?
- How do I shuffle an array randomly?
- How do I process/modify each element of an array?
- How do I select a random element from an array?
- How do I permute N elements of a list?
- How do I sort an array by (anything)?
- How do I manipulate arrays of bits?
- Why does `defined()` return true on empty arrays and hashes?
- How do I process an entire hash?
- What happens if I add or remove keys from a hash while iterating over it?
- How do I look up a hash element by value?
- How can I know how many entries are in a hash?
- How do I sort a hash (optionally by value instead of key)?
- How can I always keep my hash sorted?

- What's the difference between "delete" and "undef" with hashes?
- Why don't my tied hashes make the defined/exists distinction?
- How do I reset an `each()` operation part-way through?
- How can I get the unique keys from two hashes?
- How can I store a multidimensional array in a DBM file?
- How can I make my hash remember the order I put elements into it?
- Why does passing a subroutine an undefined element in a hash create it?
- How can I make the Perl equivalent of a C structure/C++ class/hash or array of hashes or arrays?
- How can I use a reference as a hash key?
- How do I handle binary data correctly?
- How do I determine whether a scalar is a number/whole/integer/float?
- How do I keep persistent data across program calls?
- How do I print out or copy a recursive data structure?
- How do I define methods for every class/object?
- How do I verify a credit card checksum?
- How do I pack arrays of doubles or floats for XS code?

perlfaq5: Files and Formats

I/O and the "f" issues: filehandles, flushing, formats, and footers.

- How do I flush/unbuffer an output filehandle? Why must I do this?
- How do I change one line in a file/delete a line in a file/insert a line in the middle of a file/append to the beginning of a file?
- How do I count the number of lines in a file?
- How do I make a temporary file name?
- How can I manipulate fixed-record-length files?
- How can I make a filehandle local to a subroutine? How do I pass filehandles between subroutines? How do I make an array of filehandles?
- How can I use a filehandle indirectly?
- How can I set up a footer format to be used with `write()`?
- How can I `write()` into a string?
- How can I output my numbers with commas added?
- How can I translate tildes (~) in a filename?
- How come when I open a file read-write it wipes it out?
- Why do I sometimes get an "Argument list too long" when I use `<*`?
- Is there a leak/bug in `glob()`?
- How can I open a file with a leading "" or trailing blanks?
- How can I reliably rename a file?

- How can I lock a file?
- Why can't I just open(FH, "file.lock")?
- I still don't get locking. I just want to increment the number in the file. How can I do this?
- All I want to do is append a small amount of text to the end of a file. Do I still have to use locking?
- How do I randomly update a binary file?
- How do I get a file's timestamp in perl?
- How do I set a file's timestamp in perl?
- How do I print to more than one file at once?
- How can I read in an entire file all at once?
- How can I read in a file by paragraphs?
- How can I read a single character from a file? From the keyboard?
- How can I tell whether there's a character waiting on a filehandle?
- How do I do a `tail -f` in perl?
- How do I `dup()` a filehandle in Perl?
- How do I close a file descriptor by number?
- Why can't I use "C:\temp\foo" in DOS paths? What doesn't 'C:\temp\foo.exe' work?
- Why doesn't `glob("*.*)` get all the files?
- Why does Perl let me delete read-only files? Why does `-i` clobber protected files? Isn't this a bug in Perl?
- How do I select a random line from a file?
- Why do I get weird spaces when I print an array of lines?

perlfaq6: Regular Expressions

Pattern matching and regular expressions.

- How can I hope to use regular expressions without creating illegible and unmaintainable code?
- I'm having trouble matching over more than one line. What's wrong?
- How can I pull out lines between two patterns that are themselves on different lines?
- I put a regular expression into `/` but it didn't work. What's wrong?
- How do I substitute case insensitively on the LHS while preserving case on the RHS?
- How can I make `\w` match national character sets?
- How can I match a locale-smart version of `/ [a-zA-Z] /`?
- How can I quote a variable to use in a regex?
- What is `/o` really for?
- How do I use a regular expression to strip C style comments from a file?
- Can I use Perl regular expressions to match balanced text?
- What does it mean that regexes are greedy? How can I get around it?

- How do I process each word on each line?
- How can I print out a word–frequency or line–frequency summary?
- How can I do approximate matching?
- How do I efficiently match many regular expressions at once?
- Why don't word–boundary searches with `\b` work for me?
- Why does using `$&`, `$``, or `$'` slow my program down?
- What good is `\G` in a regular expression?
- Are Perl regexes DFAs or NFAs? Are they POSIX compliant?
- What's wrong with using `grep` or `map` in a void context?
- How can I match strings with multibyte characters?
- How do I match a pattern that is supplied by the user?

perlfaq7: General Perl Language Issues

General Perl language issues that don't clearly fit into any of the other sections.

- Can I get a BNF/yacc/RE for the Perl language?
- What are all these `$_%&*` punctuation signs, and how do I know when to use them?
- Do I always/never have to quote my strings or use semicolons and commas?
- How do I skip some return values?
- How do I temporarily block warnings?
- What's an extension?
- Why do Perl operators have different precedence than C operators?
- How do I declare/create a structure?
- How do I create a module?
- How do I create a class?
- How can I tell if a variable is tainted?
- What's a closure?
- What is variable suicide and how can I prevent it?
- How can I pass/return a {Function, FileHandle, Array, Hash, Method, Regex}?
- How do I create a static variable?
- What's the difference between dynamic and lexical (static) scoping? Between `local()` and `my()`?
- How can I access a dynamic variable while a similarly named lexical is in scope?
- What's the difference between deep and shallow binding?
- Why doesn't `"my($foo) = <FILE;"` work right?
- How do I redefine a builtin function, operator, or method?
- What's the difference between calling a function as `&foo` and `foo()`?
- How do I create a switch or case statement?

- How can I catch accesses to undefined variables/functions/methods?
- Why can't a method included in this same file be found?
- How can I find out my current package?
- How can I comment out a large block of perl code?
- How do I clear a package?
- How can I use a variable as a variable name?

perlfaq8: System Interaction

Interprocess communication (IPC), control over the user-interface (keyboard, screen and pointing devices).

- How do I find out which operating system I'm running under?
- How come `exec()` doesn't return?
- How do I do fancy stuff with the keyboard/screen/mouse?
- How do I print something out in color?
- How do I read just one key without waiting for a return key?
- How do I check whether input is ready on the keyboard?
- How do I clear the screen?
- How do I get the screen size?
- How do I ask the user for a password?
- How do I read and write the serial port?
- How do I decode encrypted password files?
- How do I start a process in the background?
- How do I trap control characters/signals?
- How do I modify the shadow password file on a Unix system?
- How do I set the time and date?
- How can I `sleep()` or `alarm()` for under a second?
- How can I measure time under a second?
- How can I do an `atexit()` or `setjmp()/longjmp()`? (Exception handling)
- Why doesn't my sockets program work under System V (Solaris)? What does the error message "Protocol not supported" mean?
- How can I call my system's unique C functions from Perl?
- Where do I get the include files to do `ioctl()` or `syscall()`?
- Why do `setuid` perl scripts complain about kernel problems?
- How can I open a pipe both to and from a command?
- Why can't I get the output of a command with `system()`?
- How can I capture `STDERR` from an external command?
- Why doesn't `open()` return an error when a pipe open fails?

- What's wrong with using backticks in a void context?
- How can I call backticks without shell processing?
- Why can't my script read from STDIN after I gave it EOF (^D on Unix, ^Z on MS-DOS)?
- How can I convert my shell script to perl?
- Can I use perl to run a telnet or ftp session?
- How can I write expect in Perl?
- Is there a way to hide perl's command line from programs such as "ps"?
- I {changed directory, modified my environment} in a perl script. How come the change disappeared when I exited the script? How do I get my changes to be visible?
- How do I close a process's filehandle without waiting for it to complete?
- How do I fork a daemon process?
- How do I find out if I'm running interactively or not?
- How do I timeout a slow event?
- How do I set CPU limits?
- How do I avoid zombies on a Unix system?
- How do I use an SQL database?
- How do I make a `system()` exit on control-C?
- How do I open a file without blocking?
- How do I install a module from CPAN?
- What's the difference between `require` and `use`?
- How do I keep my own module/library directory?
- How do I add the directory my program lives in to the module/library search path?
- How do I add a directory to my include path at runtime?
- What is `socket.ph` and where do I get it?

perlfaq9: Networking

Networking, the internet, and a few on the web.

- What is the correct form of response from a CGI script?
- My CGI script runs from the command line but not the browser. (500 Server Error)
- How can I get better error messages from a CGI program?
- How do I remove HTML from a string?
- How do I extract URLs?
- How do I download a file from the user's machine? How do I open a file on another machine?
- How do I make a pop-up menu in HTML?
- How do I fetch an HTML file?
- How do I automate an HTML form submission?

- How do I decode or create those %–encodings on the web?
- How do I redirect to another page?
- How do I put a password on my web pages?
- How do I edit my .htpasswd and .htgroup files with Perl?
- How do I make sure users can't enter values into a form that cause my CGI script to do bad things?
- How do I parse a mail header?
- How do I decode a CGI form?
- How do I check a valid mail address?
- How do I decode a MIME/BASE64 string?
- How do I return the user's mail address?
- How do I send mail?
- How do I use MIME to make an attachment to a mail message?
- How do I read mail?
- How do I find out my hostname/domainname/IP address?
- How do I fetch a news article or the active newsgroups?
- How do I fetch/put an FTP file?
- How can I do RPC in Perl?

About the perlfaq documents

Where to get the perlfaq

This document is posted regularly to comp.lang.perl.announce and several other related newsgroups. It is available on many web sites: <http://www.perldoc.com/> and <http://perlfaq.cpan.org/>.

How to contribute to the perlfaq

You may mail corrections, additions, and suggestions to perlfaq-workers@perl.org. This alias should not be used to *ask* FAQs. It's for fixing the current FAQ. Send questions to the comp.lang.perl.misc newsgroup.

What will happen if you mail your Perl programming problems to the authors

Your questions will probably go unread, unless they're suggestions of new questions to add to the FAQ, in which case they should have gone to the perlfaq-workers@perl.org instead.

You should have read section 2 of this faq. There you would have learned that comp.lang.perl.misc is the appropriate place to go for free advice. If your question is really important and you require a prompt and correct answer, you should hire a consultant.

Credits

When I first began the Perl FAQ in the late 80s, I never realized it would have grown to over a hundred pages, nor that Perl would ever become so popular and widespread. This document could not have been written without the tremendous help provided by Larry Wall and the rest of the Perl Porters.

Author and Copyright Information

Copyright (c) 1997–2002 Tom Christiansen and Nathan Torkington. All rights reserved.

Bundled Distributions

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in these files are hereby placed into the public domain.

You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

Disclaimer

This information is offered in good faith and in the hope that it may be of use, but is not guaranteed to be correct, up to date, or suitable for any particular purpose whatsoever. The authors accept no liability in respect of this information or its use.

Changes

1/November/2000

A few grammatical fixes and updates implemented by John Borwick.

23/May/99

Extensive updates from the net in preparation for 5.6 release.

13/April/99

More minor touch-ups. Added new question at the end of [perlfaq7](#) on variable names within variables.

7/January/99

Small touch ups here and there. Added all questions in this document as a sort of table of contents.

22/June/98

Significant changes throughout in preparation for the 5.005 release.

24/April/97

Style and whitespace changes from Chip, new question on reading one character at a time from a terminal using POSIX from Tom.

23/April/97

Added <http://www.oasis.leo.org/perl/> to [perlfaq2](#). Style fix to [perlfaq3](#). Added floating point precision, fixed complex number arithmetic, cross-references, caveat for Text::Wrap, alternative answer for initial capitalizing, fixed incorrect regexp, added example of Tie::IxHash to [perlfaq4](#). Added example of passing and storing filehandles, added commify to [perlfaq5](#). Restored variable suicide, and added mass commenting to [perlfaq7](#). Added Net::Telnet, fixed backticks, added reader/writer pair to telnet question, added FindBin, grouped module questions together in [perlfaq8](#). Expanded caveats for the simple URL extractor, gave LWP example, added CGI security question, expanded on the mail address answer in [perlfaq9](#).

25/March/97

Added more info to the binary distribution section of [perlfaq2](#). Added Net::Telnet to [perlfaq6](#). Fixed typos in [perlfaq8](#). Added mail sending example to [perlfaq9](#). Added Merlyn's columns to [perlfaq2](#).

18/March/97

Added the DATE to the NAME section, indicating which sections have changed.

Mentioned SIGPIPE and [perlipc](#) in the forking open answer in [perlfaq8](#).

Fixed description of a regular expression in [perlfaq4](#).

17/March/97 Version

Various typos fixed throughout.

Added new question on Perl BNF on [perlfaq7](#).

Initial Release: 11/March/97

This is the initial release of version 3 of the FAQ; consequently there have been no changes since its initial release.

NAME

perlfaq1 – General Questions About Perl (\$Revision: 1.8 \$, \$Date: 2002/04/07 18:46:13 \$)

DESCRIPTION

This section of the FAQ answers very general, high-level questions about Perl.

What is Perl?

Perl is a high-level programming language with an eclectic heritage written by Larry Wall and a cast of thousands. It derives from the ubiquitous C programming language and to a lesser extent from sed, awk, the Unix shell, and at least a dozen other tools and languages. Perl's process, file, and text manipulation facilities make it particularly well-suited for tasks involving quick prototyping, system utilities, software tools, system management tasks, database access, graphical programming, networking, and world wide web programming. These strengths make it especially popular with system administrators and CGI script authors, but mathematicians, geneticists, journalists, and even managers also use Perl. Maybe you should, too.

Who supports Perl? Who develops it? Why is it free?

The original culture of the pre-populist Internet and the deeply-held beliefs of Perl's author, Larry Wall, gave rise to the free and open distribution policy of perl. Perl is supported by its users. The core, the standard Perl library, the optional modules, and the documentation you're reading now were all written by volunteers. See the personal note at the end of the README file in the perl source distribution for more details. See [perlhst](#) (new as of 5.005) for Perl's milestone releases.

In particular, the core development team (known as the Perl Porters) are a rag-tag band of highly altruistic individuals committed to producing better software for free than you could hope to purchase for money. You may snoop on pending developments via the archives at <http://www.xray.mpe.mpg.de/mailling-lists/perl5-porters/> and <http://archive.developer.com/perl5-porters@perl.org/> or the news gateway [nnntp://nnntp.perl.org/perl.perl5.porters](mailto:nnntp.perl.org/perl.perl5.porters) or its web interface at <http://nnntp.perl.org/group/perl.perl5.porters>, or read the faq at <http://perlhacker.org/p5p-faq>, or you can subscribe to the mailing list by sending perl5-porters-request@perl.org a subscription request (an empty message with no subject is fine).

While the GNU project includes Perl in its distributions, there's no such thing as "GNU Perl". Perl is not produced nor maintained by the Free Software Foundation. Perl's licensing terms are also more open than GNU software's tend to be.

You can get commercial support of Perl if you wish, although for most users the informal support will more than suffice. See the answer to "Where can I buy a commercial version of perl?" for more information.

Which version of Perl should I use?

You should definitely use version 5. Version 4 is old, limited, and no longer maintained; its last patch (4.036) was in 1992, long ago and far away. Sure, it's stable, but so is anything that's dead; in fact, perl4 had been called a dead, flea-bitten camel carcass. The most recent production release is 5.6 (although 5.005_03 is still supported). The most cutting-edge development release is 5.7. Further references to the Perl language in this document refer to the production release unless otherwise specified. There may be one or more official bug fixes by the time you read this, and also perhaps some experimental versions on the way to the next release. All releases prior to 5.004 were subject to buffer overruns, a grave security issue.

What are perl4 and perl5?

Perl4 and perl5 are informal names for different versions of the Perl programming language. It's easier to say "perl5" than it is to say "the 5.(004) release of Perl", but some people have interpreted this to mean there's a language called "perl5", which isn't the case. Perl5 is merely the popular name for the fifth major release (October 1994), while perl4 was the fourth major release (March 1991). There was also a perl1 (in January 1988), a perl2 (June 1988), and a perl3 (October 1989).

The 5.0 release is, essentially, a ground-up rewrite of the original perl source code from releases 1 through 4. It has been modularized, object-oriented, tweaked, trimmed, and optimized until it almost doesn't look like the old code. However, the interface is mostly the same, and compatibility with previous releases is very

high. See *Perl4 to Perl5 Traps in perltrap*.

To avoid the "what language is perl5?" confusion, some people prefer to simply use "perl" to refer to the latest version of perl and avoid using "perl5" altogether. It's not really that big a deal, though.

See *perlhist* for a history of Perl revisions.

What is perl6?

At The Second O'Reilly Open Source Software Convention, Larry Wall announced Perl6 development would begin in earnest. Perl6 was an oft used term for Chip Salzenberg's project to rewrite Perl in C++ named Topaz. However, Topaz provided valuable insights to the next version of Perl and its implementation, but was ultimately abandoned.

If you want to learn more about Perl6, or have a desire to help in the crusade to make Perl a better place then peruse the Perl6 developers page at <http://dev.perl.org/perl6/> and get involved.

Perl6 is not scheduled for release yet, and Perl5 will still be supported for quite awhile after its release. Do not wait for Perl6 to do whatever you need to do.

"We're really serious about reinventing everything that needs reinventing." —Larry Wall

How stable is Perl?

Production releases, which incorporate bug fixes and new functionality, are widely tested before release. Since the 5.000 release, we have averaged only about one production release per year.

Larry and the Perl development team occasionally make changes to the internal core of the language, but all possible efforts are made toward backward compatibility. While not quite all perl4 scripts run flawlessly under perl5, an update to perl should nearly never invalidate a program written for an earlier version of perl (barring accidental bug fixes and the rare new keyword).

Is Perl difficult to learn?

No, Perl is easy to start learning—and easy to keep learning. It looks like most programming languages you're likely to have experience with, so if you've ever written a C program, an awk script, a shell script, or even a BASIC program, you're already partway there.

Most tasks only require a small subset of the Perl language. One of the guiding mottos for Perl development is "there's more than one way to do it" (TMTOWTDI, sometimes pronounced "tim toady"). Perl's learning curve is therefore shallow (easy to learn) and long (there's a whole lot you can do if you really want).

Finally, because Perl is frequently (but not always, and certainly not by definition) an interpreted language, you can write your programs and test them without an intermediate compilation step, allowing you to experiment and test/debug quickly and easily. This ease of experimentation flattens the learning curve even more.

Things that make Perl easier to learn: Unix experience, almost any kind of programming experience, an understanding of regular expressions, and the ability to understand other people's code. If there's something you need to do, then it's probably already been done, and a working example is usually available for free. Don't forget the new perl modules, either. They're discussed in Part 3 of this FAQ, along with CPAN, which is discussed in Part 2.

How does Perl compare with other languages like Java, Python, REXX, Scheme, or Tcl?

Favorably in some areas, unfavorably in others. Precisely which areas are good and bad is often a personal choice, so asking this question on Usenet runs a strong risk of starting an unproductive Holy War.

Probably the best thing to do is try to write equivalent code to do a set of tasks. These languages have their own newsgroups in which you can learn about (but hopefully not argue about) them.

Some comparison documents can be found at <http://language.perl.com/versus/> if you really can't stop yourself.

Can I do [task] in Perl?

Perl is flexible and extensible enough for you to use on virtually any task, from one-line file-processing tasks to large, elaborate systems. For many people, Perl serves as a great replacement for shell scripting. For others, it serves as a convenient, high-level replacement for most of what they'd program in low-level languages like C or C++. It's ultimately up to you (and possibly your management) which tasks you'll use Perl for and which you won't.

If you have a library that provides an API, you can make any component of it available as just another Perl function or variable using a Perl extension written in C or C++ and dynamically linked into your main perl interpreter. You can also go the other direction, and write your main program in C or C++, and then link in some Perl code on the fly, to create a powerful application. See [perlembed](#).

That said, there will always be small, focused, special-purpose languages dedicated to a specific problem domain that are simply more convenient for certain kinds of problems. Perl tries to be all things to all people, but nothing special to anyone. Examples of specialized languages that come to mind include prolog and matlab.

When shouldn't I program in Perl?

When your manager forbids it—but do consider replacing them :-).

Actually, one good reason is when you already have an existing application written in another language that's all done (and done well), or you have an application language specifically designed for a certain task (e.g. prolog, make).

For various reasons, Perl is probably not well-suited for real-time embedded systems, low-level operating systems development work like device drivers or context-switching code, complex multi-threaded shared-memory applications, or extremely large applications. You'll notice that perl is not itself written in Perl.

The new, native-code compiler for Perl may eventually reduce the limitations given in the previous statement to some degree, but understand that Perl remains fundamentally a dynamically typed language, not a statically typed one. You certainly won't be chastised if you don't trust nuclear-plant or brain-surgery monitoring code to it. And Larry will sleep easier, too—Wall Street programs notwithstanding. :-)

What's the difference between "perl" and "Perl"?

One bit. Oh, you weren't talking ASCII? :-) Larry now uses "Perl" to signify the language proper and "perl" the implementation of it, i.e. the current interpreter. Hence Tom's quip that "Nothing but perl can parse Perl." You may or may not choose to follow this usage. For example, parallelism means "awk and perl" and "Python and Perl" look OK, while "awk and Perl" and "Python and perl" do not. But never write "PERL", because perl is not an acronym, apocryphal folklore and post-facto expansions notwithstanding.

Is it a Perl program or a Perl script?

Larry doesn't really care. He says (half in jest) that "a script is what you give the actors. A program is what you give the audience."

Originally, a script was a canned sequence of normally interactive commands—that is, a chat script. Something like a UUCP or PPP chat script or an expect script fits the bill nicely, as do configuration scripts run by a program at its start up, such *.cshrc* or *.ircrc*, for example. Chat scripts were just drivers for existing programs, not stand-alone programs in their own right.

A computer scientist will correctly explain that all programs are interpreted and that the only question is at what level. But if you ask this question of someone who isn't a computer scientist, they might tell you that a *program* has been compiled to physical machine code once and can then be run multiple times, whereas a *script* must be translated by a program each time it's used.

Perl programs are (usually) neither strictly compiled nor strictly interpreted. They can be compiled to a byte-code form (something of a Perl virtual machine) or to completely different languages, like C or assembly language. You can't tell just by looking at it whether the source is destined for a pure interpreter, a parse-tree interpreter, a byte-code interpreter, or a native-code compiler, so it's hard to give a definitive

answer here.

Now that "script" and "scripting" are terms that have been seized by unscrupulous or unknowing marketeers for their own nefarious purposes, they have begun to take on strange and often pejorative meanings, like "non serious" or "not real programming". Consequently, some Perl programmers prefer to avoid them altogether.

What is a JAPH?

These are the "just another perl hacker" signatures that some people sign their postings with. Randal Schwartz made these famous. About 100 of the earlier ones are available from <http://www.cpan.org/misc/japh>.

Where can I get a list of Larry Wall witticisms?

Over a hundred quips by Larry, from postings of his or source code, can be found at <http://www.cpan.org/misc/lwall-quotes.txt.gz>.

How can I convince my sysadmin/supervisor/employees to use version 5/5.6.1/Perl instead of some other language?

If your manager or employees are wary of unsupported software, or software which doesn't officially ship with your operating system, you might try to appeal to their self-interest. If programmers can be more productive using and utilizing Perl constructs, functionality, simplicity, and power, then the typical manager/supervisor/employee may be persuaded. Regarding using Perl in general, it's also sometimes helpful to point out that delivery times may be reduced using Perl compared to other languages.

If you have a project which has a bottleneck, especially in terms of translation or testing, Perl almost certainly will provide a viable, quick solution. In conjunction with any persuasion effort, you should not fail to point out that Perl is used, quite extensively, and with extremely reliable and valuable results, at many large computer software and hardware companies throughout the world. In fact, many Unix vendors now ship Perl by default. Support is usually just a news-posting away, if you can't find the answer in the *comprehensive* documentation, including this FAQ.

See <http://www.perl.org/advocacy/> for more information.

If you face reluctance to upgrading from an older version of perl, then point out that version 4 is utterly unmaintained and unsupported by the Perl Development Team. Another big sell for Perl5 is the large number of modules and extensions which greatly reduce development time for any given task. Also mention that the difference between version 4 and version 5 of Perl is like the difference between awk and C++. (Well, OK, maybe it's not quite that distinct, but you get the idea.) If you want support and a reasonable guarantee that what you're developing will continue to work in the future, then you have to run the supported version. As of January 2002 that probably means running either of the releases 5.6.1 (released in April 2001) or 5.005_03 (released in March 1999), although 5.004_05 isn't that bad if you **absolutely** need such an old version (released in April 1999) for stability reasons. Anything older than 5.004_05 shouldn't be used.

Of particular note is the massive bug hunt for buffer overflow problems that went into the 5.004 release. All releases prior to that, including perl4, are considered insecure and should be upgraded as soon as possible.

In August 2000 in all Linux distributions a new security problem was found in the optional 'suidperl' (not built or installed by default) in all the Perl branches 5.6, 5.005, and 5.004, see <http://www.cpan.org/src/5.0/sperl-2000-08-05/> Perl maintenance releases 5.6.1 and 5.8.0 have this security hole closed. Most, if not all, Linux distribution have patches for this vulnerability available, see <http://www.linuxsecurity.com/advisories/>, but the most recommendable way is to upgrade to at least Perl 5.6.1.

AUTHOR AND COPYRIGHT

Copyright (c) 1997, 1998, 1999, 2000, 2001 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples here are in the public domain. You are permitted and encouraged to use this code and any derivatives thereof in your own programs for fun or for profit as you see

fit. A simple comment in the code giving credit to the FAQ would be courteous but is not required.

NAME

perlfaq2 – Obtaining and Learning about Perl (\$Revision: 1.13 \$, \$Date: 2002/04/26 16:56:35 \$)

DESCRIPTION

This section of the FAQ answers questions about where to find source and documentation for Perl, support, and related matters.

What machines support Perl? Where do I get it?

The standard release of Perl (the one maintained by the perl development team) is distributed only in source code form. You can find this at <http://www.cpan.org/src/latest.tar.gz>, which is in a standard Internet format (a gzipped archive in POSIX tar format).

Perl builds and runs on a bewildering number of platforms. Virtually all known and current Unix derivatives are supported (Perl's native platform), as are other systems like VMS, DOS, OS/2, Windows, QNX, BeOS, OS X, MPE/iX and the Amiga.

Binary distributions for some proprietary platforms, including Apple systems, can be found <http://www.cpan.org/ports/> directory. Because these are not part of the standard distribution, they may and in fact do differ from the base Perl port in a variety of ways. You'll have to check their respective release notes to see just what the differences are. These differences can be either positive (e.g. extensions for the features of the particular platform that are not supported in the source release of perl) or negative (e.g. might be based upon a less current source release of perl).

How can I get a binary version of Perl?

If you don't have a C compiler because your vendor for whatever reasons did not include one with your system, the best thing to do is grab a binary version of gcc from the net and use that to compile perl with. CPAN only has binaries for systems that are terribly hard to get free compilers for, not for Unix systems.

Some URLs that might help you are:

```
http://www.cpan.org/ports/  
http://www.perl.com/pub/language/info/software.html
```

Someone looking for a Perl for Win16 might look to Laszlo Molnar's djgpp port in <http://www.cpan.org/ports/#msdos>, which comes with clear installation instructions. A simple installation guide for MS-DOS using Ilya Zakharevich's OS/2 port is available at <http://www.cs.ruu.nl/%7Epiet/perl5dos.html> and similarly for Windows 3.1 at <http://www.cs.ruu.nl/%7Epiet/perlwin3.html>.

I don't have a C compiler on my system. How can I compile perl?

Since you don't have a C compiler, you're doomed and your vendor should be sacrificed to the Sun gods. But that doesn't help you.

What you need to do is get a binary version of gcc for your system first. Consult the Usenet FAQs for your operating system for information on where to get such a binary version.

I copied the Perl binary from one machine to another, but scripts don't work.

That's probably because you forgot libraries, or library paths differ. You really should build the whole distribution on the machine it will eventually live on, and then type `make install`. Most other approaches are doomed to failure.

One simple way to check that things are in the right place is to print out the hard-coded @INC that perl looks through for libraries:

```
% perl -e 'print join("\n",@INC)'
```

If this command lists any paths that don't exist on your system, then you may need to move the appropriate libraries to these locations, or create symbolic links, aliases, or shortcuts appropriately. @INC is also printed as part of the output of

```
% perl -V
```

You might also want to check out [How do I keep my own module/library directory? in perlfaq8](#).

I grabbed the sources and tried to compile but gdbm/dynamic loading/malloc/linking/... failed. How do I make it work?

Read the *INSTALL* file, which is part of the source distribution. It describes in detail how to cope with most idiosyncrasies that the Configure script can't work around for any given system or architecture.

What modules and extensions are available for Perl? What is CPAN? What does CPAN/src/... mean?

CPAN stands for Comprehensive Perl Archive Network, a ~700mb archive replicated on nearly 200 machines all over the world. CPAN contains source code, non-native ports, documentation, scripts, and many third-party modules and extensions, designed for everything from commercial database interfaces to keyboard/screen control to web walking and CGI scripts. The master web site for CPAN is <http://www.cpan.org/> and there is the CPAN Multiplexer at <http://www.cpan.org/CPAN.html> which will choose a mirror near you via DNS. See <http://www.perl.com/CPAN> (without a slash at the end) for how this process works. Also, <http://mirror.cpan.org/> has a nice interface to the <http://www.cpan.org/MIRRORED.BY> mirror directory.

See the CPAN FAQ at <http://www.cpan.org/misc/cpan-faq.html> for answers to the most frequently asked questions about CPAN including how to become a mirror.

CPAN/path/... is a naming convention for files available on CPAN sites. CPAN indicates the base directory of a CPAN mirror, and the rest of the path is the path from that directory to the file. For instance, if you're using <ftp://ftp.funet.fi/pub/languages/perl/CPAN> as your CPAN site, the file `CPAN/misc/japh` is downloadable as <ftp://ftp.funet.fi/pub/languages/perl/CPAN/misc/japh>.

Considering that there are close to two thousand existing modules in the archive, one probably exists to do nearly anything you can think of. Current categories under `CPAN/modules/by-category/` include Perl core modules; development support; operating system interfaces; networking, devices, and interprocess communication; data type utilities; database interfaces; user interfaces; interfaces to other languages; filenames, file systems, and file locking; internationalization and locale; world wide web support; server and daemon utilities; archiving and compression; image manipulation; mail and news; control flow utilities; filehandle and I/O; Microsoft Windows modules; and miscellaneous modules.

See <http://www.cpan.org/modules/00modlist.long.html> or <http://search.cpan.org/> for a more complete list of modules by category.

CPAN is not affiliated with O'Reilly and Associates.

Is there an ISO or ANSI certified version of Perl?

Certainly not. Larry expects that he'll be certified before Perl is.

Where can I get information on Perl?

The complete Perl documentation is available with the Perl distribution. If you have Perl installed locally, you probably have the documentation installed as well: type `man perl` if you're on a system resembling Unix. This will lead you to other important man pages, including how to set your `$MANPATH`. If you're not on a Unix system, access to the documentation will be different; for example, documentation might only be in HTML format. All proper Perl installations have fully-accessible documentation.

You might also try `perldoc perl` in case your system doesn't have a proper `man` command, or it's been misinstalled. If that doesn't work, try looking in `/usr/local/lib/perl5/pod` for documentation.

If all else fails, consult <http://perldoc.cpan.org/> or <http://www.perldoc.com/> both offer the complete documentation in html format.

Many good books have been written about Perl—see the section below for more details.

Tutorial documents are included in current or upcoming Perl releases include [perltoot](#) for objects or [perlboot](#) for a beginner's approach to objects, [perlopentut](#) for file opening semantics, [perlrefut](#) for managing

references, [perlretut](#) for regular expressions, [perlthrtut](#) for threads, [perldebtut](#) for debugging, and [perlxstut](#) for linking C and Perl together. There may be more by the time you read this. The following URLs might also be of assistance:

```
http://perldoc.cpan.org/
http://www.perldoc.com/
http://reference.perl.com/query.cgi?tutorials
http://bookmarks.cpan.org/search.cgi?cat=Training%2FTutorials
```

What are the Perl newsgroups on Usenet? Where do I post questions?

The now defunct comp.lang.perl newsgroup has been superseded by the following groups:

comp.lang.perl.announce	Moderated announcement group
comp.lang.perl.misc	Very busy group about Perl in general
comp.lang.perl.moderated	Moderated discussion group
comp.lang.perl.modules	Use and development of Perl modules
comp.lang.perl.tk	Using Tk (and X) from Perl
comp.infosystems.www.authoring.cgi	Writing CGI scripts for the Web.

There is also a Usenet gateway to Perl mailing lists sponsored by perl.org at [nntp://nntp.perl.org](http://nntp.perl.org), a web interface to the same lists at <http://nntp.perl.org/group/> and these lists are also available under the perl.* hierarchy at <http://groups.google.com>. Other groups are listed at <http://lists.perl.org/> (also known as <http://lists.cpan.org/>).

A nice place to ask questions is the PerlMonks site, <http://www.perlmonks.org/>

Note that none of the above are supposed to write your code for you: asking questions about particular problems or general advice is fine, but asking someone to write your code for free is not very cool.

Where should I post source code?

You should post source code to whichever group is most appropriate, but feel free to cross-post to comp.lang.perl.misc. If you want to cross-post to alt.sources, please make sure it follows their posting standards, including setting the Followup-To header line to NOT include alt.sources; see their FAQ (<http://www.faqs.org/faqs/alt-sources-intro/>) for details.

If you're just looking for software, first use Google (<http://www.google.com>), Google's usenet search interface (<http://groups.google.com>), and CPAN Search (<http://search.cpan.org>). This is faster and more productive than just posting a request.

Perl Books

A number of books on Perl and/or CGI programming are available. A few of these are good, some are OK, but many aren't worth your money. Tom Christiansen maintains a list of these books, some with extensive reviews, at <http://www.perl.com/perl/critiques/index.html>.

The incontestably definitive reference book on Perl, written by the creator of Perl, is now (July 2000) in its third edition:

```
Programming Perl (the "Camel Book"):
  by Larry Wall, Tom Christiansen, and Jon Orwant
  0-596-00027-8 [3rd edition July 2000]
  http://www.oreilly.com/catalog/ppperl3/
  (English, translations to several languages are also available)
```

The companion volume to the Camel containing thousands of real-world examples, mini-tutorials, and complete programs is:

```
The Perl Cookbook (the "Ram Book"):
  by Tom Christiansen and Nathan Torkington,
  with Foreword by Larry Wall
  ISBN 1-56592-243-3 [1st Edition August 1998]
```

<http://perl.oreilly.com/cookbook/>

If you're already a seasoned programmer, then the Camel Book might suffice for you to learn Perl from. If you're not, check out the Llama book:

Learning Perl (the "Llama Book")
by Randal L. Schwartz and Tom Phoenix
ISBN 0-596-00132-0 [3rd edition July 2001]
<http://www.oreilly.com/catalog/lperl3/>

If you're not an accidental programmer, but a more serious and possibly even degreed computer scientist who doesn't need as much hand-holding as we try to provide in the Llama, please check out the delightful book

Perl: The Programmer's Companion
by Nigel Chapman
ISBN 0-471-97563-X [1997, 3rd printing Spring 1998]
<http://www.wiley.com/compbooks/catalog/97563-X.htm>
<http://www.wiley.com/compbooks/chapman/perl/perltpc.html> (errata etc)

If you are more at home in Windows the following is available (though unfortunately rather dated).

Learning Perl on Win32 Systems (the "Gecko Book")
by Randal L. Schwartz, Erik Olson, and Tom Christiansen,
with foreword by Larry Wall
ISBN 1-56592-324-3 [1st edition August 1997]
<http://www.oreilly.com/catalog/lperlwin/>

Addison-Wesley (<http://www.awlonline.com/>) and Manning (<http://www.manning.com/>) are also publishers of some fine Perl books such as *Object Oriented Programming with Perl* by Damian Conway and *Network Programming with Perl* by Lincoln Stein.

An excellent technical book discounter is Bookpool at <http://www.bookpool.com/> where a 30% discount or more is not unusual.

What follows is a list of the books that the FAQ authors found personally useful. Your mileage may (but, we hope, probably won't) vary.

Recommended books on (or mostly on) Perl follow.

References

Programming Perl
by Larry Wall, Tom Christiansen, and Jon Orwant
ISBN 0-596-00027-8 [3rd edition July 2000]
<http://www.oreilly.com/catalog/ppperl3/>

Perl 5 Pocket Reference
by Johan Vromans
ISBN 0-596-00032-4 [3rd edition May 2000]
<http://www.oreilly.com/catalog/perlpr3/>

Perl in a Nutshell
by Ellen Siever, Stephan Spainhour, and Nathan Patwardhan
ISBN 1-56592-286-7 [1st edition December 1998]
<http://www.oreilly.com/catalog/perlmut/>

Tutorials

Elements of Programming with Perl
by Andrew L. Johnson
ISBN 1-884777-80-5 [1st edition October 1999]
<http://www.manning.com/Johnson/>

Learning Perl

by Randal L. Schwartz and Tom Phoenix
ISBN 0-596-00132-0 [3rd edition July 2001]
<http://www.oreilly.com/catalog/lperl3/>

Learning Perl on Win32 Systems

by Randal L. Schwartz, Erik Olson, and Tom Christiansen,
with foreword by Larry Wall
ISBN 1-56592-324-3 [1st edition August 1997]
<http://www.oreilly.com/catalog/lperlwin/>

Perl: The Programmer's Companion

by Nigel Chapman
ISBN 0-471-97563-X [1997, 3rd printing Spring 1998]
<http://www.wiley.com/compbooks/catalog/97563-X.htm>
<http://www.wiley.com/compbooks/chapman/perl/perltpc.html> (errata etc)

Cross-Platform Perl

by Eric Foster-Johnson
ISBN 1-55851-483-X [2nd edition September 2000]
<http://www.pconline.com/~erc/perlbook.htm>

MacPerl: Power and Ease

by Vicki Brown and Chris Nandor,
with foreword by Matthias Neeracher
ISBN 1-881957-32-2 [1st edition May 1998]
http://www.macperl.com/ptf_book/

Task-Oriented

The Perl Cookbook

by Tom Christiansen and Nathan Torkington
with foreword by Larry Wall
ISBN 1-56592-243-3 [1st edition August 1998]
<http://www.oreilly.com/catalog/cookbook/>

Effective Perl Programming

by Joseph Hall
ISBN 0-201-41975-0 [1st edition 1998]
<http://www.awl.com/>

Special Topics

Mastering Regular Expressions

by Jeffrey E. F. Friedl
ISBN 1-56592-257-3 [1st edition January 1997]
<http://www.oreilly.com/catalog/regex/>

Network Programming with Perl

by Lincoln Stein
ISBN 0-201-61571-1 [1st edition 2001]
<http://www.awlonline.com/>

Object Oriented Perl

Damian Conway
with foreword by Randal L. Schwartz
ISBN 1-884777-79-1 [1st edition August 1999]
<http://www.manning.com/Conway/>

Data Munging with Perl

Dave Cross

ISBN 1-930110-00-6 [1st edition 2001]
<http://www.manning.com/cross>

Mastering Perl/Tk
by Steve Lidie and Nancy Walsh
ISBN 1-56592-716-8 [1st edition January 2002]
<http://www.oreilly.com/catalog/mastperlTk/>

Perl in Magazines

The first (and for a long time, only) periodical devoted to All Things Perl, *The Perl Journal* contains tutorials, demonstrations, case studies, announcements, contests, and much more. *TPJ* has columns on web development, databases, Win32 Perl, graphical programming, regular expressions, and networking, and sponsors the Obfuscated Perl Contest and the Perl Poetry Contests. As of mid-2001, the dead tree version of *TPJ* will be published as a quarterly supplement of *SysAdmin* magazine (<http://www.sysadminmag.com/>) For more details on *TPJ*, see <http://www.tpj.com/>

Beyond this, magazines that frequently carry quality articles on Perl are *The Perl Review* (<http://www.theperlreview.com/>), *Unix Review* (<http://www.unixreview.com/>), *Linux Magazine* (<http://www.linuxmagazine.com/>), and Usenix's newsletter/magazine to its members, *login:* (<http://www.usenix.org/>)

The Perl columns of Randal L. Schwartz are available on the web at <http://www.stonehenge.com/merlyn/WebTechniques/> , <http://www.stonehenge.com/merlyn/UnixReview/> , and <http://www.stonehenge.com/merlyn/LinuxMag/> .

Perl on the Net: FTP and WWW Access

To get the best performance, pick a site from the list below and use it to grab the complete list of mirror sites which is at `/CPAN/MIRRORED.BY` or at <http://mirror.cpan.org/> . From there you can find the quickest site for you. Remember, the following list is *not* the complete list of CPAN mirrors (the complete list contains 165 sites as of January 2001):

```
http://www.cpan.org/  
http://www.perl.com/CPAN/  
http://download.sourceforge.net/mirrors/CPAN/  
ftp://ftp.digital.com/pub/plan/perl/CPAN/  
ftp://ftp.flirble.org/pub/languages/perl/CPAN/  
ftp://ftp.uvsq.fr/pub/perl/CPAN/  
ftp://ftp.funet.fi/pub/languages/perl/CPAN/  
ftp://ftp.dti.ad.jp/pub/lang/CPAN/  
ftp://cpan.if.usp.br/pub/mirror/CPAN/
```

One may also use `xx.cpan.org` where "xx" is the 2-letter country code for your domain; e.g. Australia would use `au.cpan.org`. [Note: This only applies to countries that host at least one mirror.]

What mailing lists are there for Perl?

Most of the major modules (Tk, CGI, libwww-perl) have their own mailing lists. Consult the documentation that came with the module for subscription information.

A comprehensive list of Perl related mailing lists can be found at:

<http://lists.cpan.org/>

(also visible as <http://lists.perl.org/>)

Archives of comp.lang.perl.misc

The Google search engine now carries archived and searchable newsgroup content.

<http://groups.google.com/groups?group=comp.lang.perl.misc>

If you have a question, you can be sure someone has already asked the same question at some point on c.l.p.m. It requires some time and patience to sift through all the content but often you will find the answer

you seek.

Where can I buy a commercial version of Perl?

In a real sense, Perl already *is* commercial software: it has a license that you can grab and carefully read to your manager. It is distributed in releases and comes in well-defined packages. There is a very large user community and an extensive literature. The comp.lang.perl.* newsgroups and several of the mailing lists provide free answers to your questions in near real-time. Perl has traditionally been supported by Larry, scores of software designers and developers, and myriad programmers, all working for free to create a useful thing to make life better for everyone.

However, these answers may not suffice for managers who require a purchase order from a company whom they can sue should anything go awry. Or maybe they need very serious hand-holding and contractual obligations. Shrink-wrapped CDs with Perl on them are available from several sources if that will help. For example, many Perl books include a distribution of Perl, as do the O'Reilly Perl Resource Kits (in both the Unix flavor and in the proprietary Microsoft flavor); the free Unix distributions also all come with Perl.

Alternatively, you can purchase commercial incidence based support through the Perl Clinic. The following is a commercial from them:

"The Perl Clinic is a commercial Perl support service operated by ActiveState Tool Corp. and The Ingram Group. The operators have many years of in-depth experience with Perl applications and Perl internals on a wide range of platforms.

"Through our group of highly experienced and well-trained support engineers, we will put our best effort into understanding your problem, providing an explanation of the situation, and a recommendation on how to proceed."

Contact The Perl Clinic at

`www.PperlClinic.com`

North America Pacific Standard Time (GMT-8)

Tel: 1 604 606-4611 hours 8am-6pm

Fax: 1 604 606-4640

Europe (GMT)

Tel: 00 44 1483 862814

Fax: 00 44 1483 862801

See also www.perl.com for updates on tutorials, training, and support.

Where do I send bug reports?

If you are reporting a bug in the perl interpreter or the modules shipped with Perl, use the *perlbug* program in the Perl distribution or mail your report to perlbug@perl.org.

If you are posting a bug with a non-standard port (see the answer to "What platforms is Perl available for?"), a binary distribution, or a non-standard module (such as Tk, CGI, etc), then please see the documentation that came with it to determine the correct place to post bugs.

Read the *perlbug(1)* man page (*perl5.004* or later) for more information.

What is perl.com? Perl Mongers? pm.org? perl.org? cpan.org?

The Perl Home Page at <http://www.perl.com/> is currently hosted by The O'Reilly Network, a subsidiary of O'Reilly and Associates.

Perl Mongers is an advocacy organization for the Perl language which maintains the web site <http://www.perl.org/> as a general advocacy site for the Perl language.

Perl Mongers uses the *pm.org* domain for services related to Perl user groups, including the hosting of mailing lists and web sites. See the Perl user group web site at <http://www.pm.org/> for more information about joining, starting, or requesting services for a Perl user group.

Perl Mongers also maintain the perl.org domain to provide general support services to the Perl community, including the hosting of mailing lists, web sites, and other services. The web site <http://www.perl.org/> is a general advocacy site for the Perl language, and there are many other sub-domains for special topics, such as

```
http://bugs.perl.org/  
http://history.perl.org/  
http://lists.perl.org/  
http://use.perl.org/
```

<http://www.cpan.org/> is the Comprehensive Perl Archive Network, a replicated worldwide repository of Perl software, see the *What is CPAN?* question earlier in this document.

AUTHOR AND COPYRIGHT

Copyright (c) 1997–2001 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples here are in the public domain. You are permitted and encouraged to use this code and any derivatives thereof in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit to the FAQ would be courteous but is not required.

NAME

perlfaq3 – Programming Tools (\$Revision: 1.22 \$, \$Date: 2002/05/06 13:11:13 \$)

DESCRIPTION

This section of the FAQ answers questions related to programmer tools and programming support.

How do I do (anything)?

Have you looked at CPAN (see [perlfaq2](#))? The chances are that someone has already written a module that can solve your problem. Have you read the appropriate manpages? Here's a brief index:

Basics	perldata , perlvar , perlsyn , perlop , perlsub
Execution	perlrun , perldebug
Functions	perlfunc
Objects	perlref , perlmod , perlobj , perltie
Data Structures	perlref , perllo , perldsc
Modules	perlmod , perlmodlib , perlsub
Regexes	perlre , perlfunc , perlop , perllocale
Moving to perl5	perltrap , perl
Linking w/C	perlxs , perlcall , perlguts , perlembed
Various	http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz (not a man-page but still useful, a collection of various essays on Perl techniques)

A crude table of contents for the Perl manpage set is found in [perltoc](#).

How can I use Perl interactively?

The typical approach uses the Perl debugger, described in the [perldebug\(1\)](#) manpage, on an “empty” program, like this:

```
perl -de 42
```

Now just type in any legal Perl code, and it will be immediately evaluated. You can also examine the symbol table, get stack backtraces, check variable values, set breakpoints, and other operations typically found in symbolic debuggers.

Is there a Perl shell?

In general, not yet. There is psh available at

```
http://www.focusresearch.com/gregor/psh
```

Which includes the following description:

```
The Perl Shell is a shell that combines the interactive nature
of a Unix shell with the power of Perl. The goal is to eventually
have a full featured shell that behaves as expected for normal
shell activity. But, the Perl Shell will use Perl syntax and
functionality for control-flow statements and other things.
```

The `Shell.pm` module (distributed with Perl) makes Perl try commands which aren't part of the Perl language as shell commands. `perlsh` from the source distribution is simplistic and uninteresting, but may still be what you want.

How do I debug my Perl programs?

Have you tried `use warnings` or `used -w`? They enable warnings to detect dubious practices.

Have you tried `use strict`? It prevents you from using symbolic references, makes you predeclare any subroutines that you call as bare words, and (probably most importantly) forces you to predeclare your variables with `my`, `our`, or `use vars`.

Did you check the return values of each and every system call? The operating system (and thus Perl) tells

you whether they worked, and if not why.

```
open(FH, "> /etc/cantwrite")
  or die "Couldn't write to /etc/cantwrite: $!\n";
```

Did you read [perltrap](#)? It's full of gotchas for old and new Perl programmers and even has sections for those of you who are upgrading from languages like *awk* and *C*.

Have you tried the Perl debugger, described in [perldebug](#)? You can step through your program and see what it's doing and thus work out why what it's doing isn't what it should be doing.

How do I profile my Perl programs?

You should get the `Devel::DProf` module from the standard distribution (or separately on CPAN) and also use `Benchmark.pm` from the standard distribution. The `Benchmark` module lets you time specific portions of your code, while `Devel::DProf` gives detailed breakdowns of where your code spends its time.

Here's a sample use of `Benchmark`:

```
use Benchmark;

@junk = `cat /etc/motd`;
$count = 10_000;

timethese($count, {
    'map' => sub { my @a = @junk;
                  map { s/a/b/ } @a;
                  return @a
                },
    'for' => sub { my @a = @junk;
                  local $_;
                  for (@a) { s/a/b/ };
                  return @a },
});
```

This is what it prints (on one machine—your results will be dependent on your hardware, operating system, and the load on your machine):

```
Benchmark: timing 10000 iterations of for, map...
   for:  4 secs ( 3.97 usr  0.01 sys =  3.98 cpu)
   map:  6 secs ( 4.97 usr  0.00 sys =  4.97 cpu)
```

Be aware that a good benchmark is very hard to write. It only tests the data you give it and proves little about the differing complexities of contrasting algorithms.

How do I cross-reference my Perl programs?

The `B::Xref` module can be used to generate cross-reference reports for Perl programs.

```
perl -MO=Xref[,OPTIONS] scriptname.plx
```

Is there a pretty-printer (formatter) for Perl?

`Perltidy` is a Perl script which indents and reformats Perl scripts to make them easier to read by trying to follow the rules of the [perlstyle](#). If you write Perl scripts, or spend much time reading them, you will probably find it useful. It is available at <http://perltidy.sourceforge.net>

Of course, if you simply follow the guidelines in [perlstyle](#), you shouldn't need to reformat. The habit of formatting your code as you write it will help prevent bugs. Your editor can and should help you with this. The `perl-mode` or newer `cperl-mode` for `emacs` can provide remarkable amounts of help with most (but not all) code, and even less programmable editors can provide significant assistance. Tom Christiansen and many other VI users swear by the following settings in `vi` and its clones:

```
set ai sw=4
map! ^O {^M}^[O^T
```

Put that in your *.exrc* file (replacing the caret characters with control characters) and away you go. In insert mode, ^T is for indenting, ^D is for undenting, and ^O is for blockdenting— as it were. A more complete example, with comments, can be found at <http://www.cpan.org/authors/id/TOMC/scripts/toms.exrc.gz>

The a2ps <http://www-inf.enst.fr/%7Edemaille/a2ps/black+white.ps> does lots of things related to generating nicely printed output of documents, as does *enscript* at <http://people.ssh.fi/mtr/genscript/>.

Is there a ctags for Perl?

Recent versions of ctags do much more than older versions did. EXUBERANT CTAGS is available from <http://ctags.sourceforge.net/> and does a good job of making tags files for perl code.

There is also a simple one at <http://www.cpan.org/authors/id/TOMC/scripts/ptags.gz> which may do the trick. It can be easy to hack this into what you want.

Is there an IDE or Windows Perl Editor?

Perl programs are just plain text, so any editor will do.

If you're on Unix, you already have an IDE—Unix itself. The UNIX philosophy is the philosophy of several small tools that each do one thing and do it well. It's like a carpenter's toolbox.

If you want an IDE, check the following:

Komodo

ActiveState's cross-platform (as of April 2001 Windows and Linux), multi-language IDE has Perl support, including a regular expression debugger and remote debugging (<http://www.ActiveState.com/Products/Komodo/index.html>). (Visual Perl, a Visual Studio.NET plug-in is currently (early 2001) in beta (<http://www.ActiveState.com/Products/VisualPerl/index.html>)).

The Object System

(http://www.castlelink.co.uk/object_system/) is a Perl web applications development IDE, apparently for any platform that runs Perl.

Open Perl IDE

(<http://open-perl-ide.sourceforge.net/>) Open Perl IDE is an integrated development environment for writing and debugging Perl scripts with ActiveState's ActivePerl distribution under Windows 95/98/NT/2000.

PerlBuilder

(<http://www.solutionsoft.com/perl.htm>) is an integrated development environment for Windows that supports Perl development.

visiPerl+

(<http://helpconsulting.net/visiperl/>) From Help Consulting, for Windows.

OptiPerl

(<http://www.optiperl.com/>) is a Windows IDE with simulated CGI environment, including debugger and syntax highlighting editor.

For Windows there's also the

CodeMagicCD

(<http://www.codemagiccd.com/>) Collection of various programming tools for Windows: Perl (5.005_03), TclTk, Python, GNU programming tools, REBOL, wxWindows toolkit, the MinGW GNU C/C++ compiler, DJGPP GNU C/C++ compiler, Cint C interpreter, YaBasic.

For editors: if you're on Unix you probably have vi or a vi clone already, and possibly an emacs too, so you may not need to download anything. In any emacs the *cperl-mode* (*M-x cperl-mode*) gives you perhaps the best available Perl editing mode in any editor.

If you are using Windows, you can use any editor that lets you work with plain text, such as NotePad or WordPad. Word processors, such as Microsoft Word or WordPerfect, typically do not work since they insert all sorts of behind-the-scenes information, although some allow you to save files as "Text Only". You can also download text editors designed specifically for programming, such as Textpad (<http://www.textpad.com/>) and UltraEdit (<http://www.ultraedit.com/>), among others.

If you are using Mac OS, the same concerns apply. MacPerl (for Classic environments) comes with a simple editor. Popular external editors are BBEdit (<http://www.bbedit.com/>) or Alpha (<http://www.kelehers.org/alpha/>). Mac OS X users can use Unix editors as well.

GNU Emacs

<http://www.gnu.org/software/emacs/windows/ntemacs.html>

MicroEMACS

<http://members.nbci.com/uemacs/>

XEmacs

<http://www.xemacs.org/Download/index.html>

or a vi clone such as

Elvis

<ftp://ftp.cs.pdx.edu/pub/elvis/> <http://www.fh-wedel.de/elvis/>

Vile <http://vile.cx/>

Vim <http://www.vim.org/>

win32: <http://www.cs.vu.nl/%7Etmgil/vi.html>

For vi lovers in general, Windows or elsewhere:

<http://www.thomer.com/thomer/vi/vi.html>

nvi (<http://www.bostic.com/vi/> , available from CPAN in `src/misc/`) is yet another vi clone, unfortunately not available for Windows, but in UNIX platforms you might be interested in trying it out, firstly because strictly speaking it is not a vi clone, it is the real vi, or the new incarnation of it, and secondly because you can embed Perl inside it to use Perl as the scripting language. nvi is not alone in this, though: at least also vim and vile offer an embedded Perl.

The following are Win32 multilanguage editor/IDESs that support Perl:

Codewright

<http://www.starbase.com/>

MultiEdit

<http://www.MultiEdit.com/>

SlickEdit

<http://www.slickedit.com/>

There is also a toypedit Text widget based editor written in Perl that is distributed with the Tk module on CPAN. The ptkdb (<http://world.std.com/~aep/ptkdb/>) is a Perl/tk based debugger that acts as a development environment of sorts. Perl Composer (<http://perlcomposer.sourceforge.net/vperl.html>) is an IDE for Perl/Tk GUI creation.

In addition to an editor/IDE you might be interested in a more powerful shell environment for Win32. Your options include

Bash

from the Cygwin package (<http://sources.redhat.com/cygwin/>)

Ksh from the MKS Toolkit (<http://www.mks.com/>), or the Bourne shell of the U/WIN environment (<http://www.research.att.com/sw/tools/uwin/>)

Tcsh

<ftp://ftp.astron.com/pub/tcsh/> , see also <http://www.primite.wisc.edu/software/csh-tcsh-book/>

Zsh <ftp://ftp.blarg.net/users/amol/zsh/> , see also <http://www.zsh.org/>

MKS and U/WIN are commercial (U/WIN is free for educational and research purposes), Cygwin is covered by the GNU Public License (but that shouldn't matter for Perl use). The Cygwin, MKS, and U/WIN all contain (in addition to the shells) a comprehensive set of standard UNIX toolkit utilities.

If you're transferring text files between Unix and Windows using FTP be sure to transfer them in ASCII mode so the ends of lines are appropriately converted.

On Mac OS the MacPerl Application comes with a simple 32k text editor that behaves like a rudimentary IDE. In contrast to the MacPerl Application the MPW Perl tool can make use of the MPW Shell itself as an editor (with no 32k limit).

BBEdit and BBEdit Lite

are text editors for Mac OS that have a Perl sensitivity mode (<http://web.barebones.com/>).

Alpha

is an editor, written and extensible in Tcl, that nonetheless has built in support for several popular markup and programming languages including Perl and HTML (<http://alpha.olm.net/>).

Pepper and Pe are programming language sensitive text editors for Mac OS X and BeOS respectively (<http://www.hekkelman.com/>).

Where can I get Perl macros for vi?

For a complete version of Tom Christiansen's vi configuration file, see http://www.cpan.org/authors/Tom_Christiansen/scripts/toms.exrc.gz , the standard benchmark file for vi emulators. The file runs best with nvi, the current version of vi out of Berkeley, which incidentally can be built with an embedded Perl interpreter—see <http://www.cpan.org/src/misc/> .

Where can I get perl-mode for emacs?

Since Emacs version 19 patchlevel 22 or so, there have been both a perl-mode.el and support for the Perl debugger built in. These should come with the standard Emacs 19 distribution.

In the Perl source directory, you'll find a directory called "emacs", which contains a cperl-mode that color-codes keywords, provides context-sensitive help, and other nifty things.

Note that the perl-mode of emacs will have fits with "main`foo" (single quote), and mess up the indentation and highlighting. You are probably using "main::foo" in new Perl code anyway, so this shouldn't be an issue.

How can I use curses with Perl?

The Curses module from CPAN provides a dynamically loadable object module interface to a curses library. A small demo can be found at the directory http://www.cpan.org/authors/Tom_Christiansen/scripts/rep ; this program repeats a command and updates the screen as needed, rendering **rep ps axu** similar to **top**.

How can I use X or Tk with Perl?

Tk is a completely Perl-based, object-oriented interface to the Tk toolkit that doesn't force you to use Tcl just to get at Tk. Sx is an interface to the Athena Widget set. Both are available from CPAN. See the directory http://www.cpan.org/modules/by-category/08_User_Interfaces/

Invaluable for Perl/Tk programming are the Perl/Tk FAQ at <http://w4.lns.cornell.edu/%7Epvh/ptk/ptkTOC.html> , the Perl/Tk Reference Guide available at http://www.cpan.org/authors/Stephen_O_Lidie/ , and the online manpages at <http://www-users.cs.umn.edu/%7Eamundson/perl/perlTk/toc.html> .

How can I generate simple menus without using CGI or Tk?

The <http://www.cpan.org/authors/id/SKUNZ/perlmenu.v4.0.tar.gz> module, which is curses-based, can help with this.

How can I make my Perl program run faster?

The best way to do this is to come up with a better algorithm. This can often make a dramatic difference. Jon Bentley's book "Programming Pearls" (that's not a misspelling!) has some good tips on optimization, too. Advice on benchmarking boils down to: benchmark and profile to make sure you're optimizing the right part, look for better algorithms instead of microtuning your code, and when all else fails consider just buying faster hardware. You will probably want to read the answer to the earlier question "How do I profile my Perl programs?" if you haven't done so already.

A different approach is to autoload seldom-used Perl code. See the `AutoSplit` and `AutoLoader` modules in the standard distribution for that. Or you could locate the bottleneck and think about writing just that part in C, the way we used to take bottlenecks in C code and write them in assembler. Similar to rewriting in C, modules that have critical sections can be written in C (for instance, the `PDL` module from CPAN).

In some cases, it may be worth it to use the backend compiler to produce byte code (saving compilation time) or compile into C, which will certainly save compilation time and sometimes a small amount (but not much) execution time. See the question about compiling your Perl programs for more on the compiler—the wins aren't as obvious as you'd hope.

If you're currently linking your perl executable to a shared *libc.so*, you can often gain a 10–25% performance benefit by rebuilding it to link with a static *libc.a* instead. This will make a bigger perl executable, but your Perl programs (and programmers) may thank you for it. See the *INSTALL* file in the source distribution for more information.

Unsubstantiated reports allege that Perl interpreters that use *sfio* outperform those that don't (for I/O intensive applications). To try this, see the *INSTALL* file in the source distribution, especially the "Selecting File I/O mechanisms" section.

The undump program was an old attempt to speed up your Perl program by storing the already-compiled form to disk. This is no longer a viable option, as it only worked on a few architectures, and wasn't a good solution anyway.

How can I make my Perl program take less memory?

When it comes to time-space tradeoffs, Perl nearly always prefers to throw memory at a problem. Scalars in Perl use more memory than strings in C, arrays take more than that, and hashes use even more. While there's still a lot to be done, recent releases have been addressing these issues. For example, as of 5.004, duplicate hash keys are shared amongst all hashes using them, so require no reallocation.

In some cases, using `substr()` or `vec()` to simulate arrays can be highly beneficial. For example, an array of a thousand booleans will take at least 20,000 bytes of space, but it can be turned into one 125-byte bit vector—a considerable memory savings. The standard `Tie::SubstrHash` module can also help for certain types of data structure. If you're working with specialist data structures (matrices, for instance) modules that implement these in C may use less memory than equivalent Perl modules.

Another thing to try is learning whether your Perl was compiled with the system `malloc` or with Perl's builtin `malloc`. Whichever one it is, try using the other one and see whether this makes a difference. Information about `malloc` is in the *INSTALL* file in the source distribution. You can find out whether you are using perl's `malloc` by typing `perl -V:usemymalloc`.

Of course, the best way to save memory is to not do anything to waste it in the first place. Good programming practices can go a long way toward this:

- Don't slurp!

Don't read an entire file into memory if you can process it line by line. Or more concretely, use a loop like this:

```
#
# Good Idea
#
while (<FILE>) {
    # ...
}
```

instead of this:

```
#
# Bad Idea
#
@data = <FILE>;
foreach (@data) {
    # ...
}
```

When the files you're processing are small, it doesn't much matter which way you do it, but it makes a huge difference when they start getting larger.

- Use map and grep selectively

Remember that both map and grep expect a LIST argument, so doing this:

```
@wanted = grep {/pattern/} <FILE>;
```

will cause the entire file to be slurped. For large files, it's better to loop:

```
while (<FILE>) {
    push(@wanted, $_) if /pattern/;
}
```

- Avoid unnecessary quotes and stringification

Don't quote large strings unless absolutely necessary:

```
my $copy = "$large_string";
```

makes 2 copies of \$large_string (one for \$copy and another for the quotes), whereas

```
my $copy = $large_string;
```

only makes one copy.

Ditto for stringifying large arrays:

```
{
    local $, = "\n";
    print @big_array;
}
```

is much more memory-efficient than either

```
print join "\n", @big_array;
```

or

```
{
    local $" = "\n";
    print "@big_array";
}
```

- Pass by reference

Pass arrays and hashes by reference, not by value. For one thing, it's the only way to pass multiple lists or hashes (or both) in a single call/return. It also avoids creating a copy of all the contents. This

requires some judgment, however, because any changes will be propagated back to the original data. If you really want to mangle (er, modify) a copy, you'll have to sacrifice the memory needed to make one.

- Tie large variables to disk.

For "big" data stores (i.e. ones that exceed available memory) consider using one of the DB modules to store it on disk instead of in RAM. This will incur a penalty in access time, but that's probably better than causing your hard disk to thrash due to massive swapping.

Is it unsafe to return a pointer to local data?

No, Perl's garbage collection system takes care of this.

```
sub makeone {
    my @a = ( 1 .. 10 );
    return \@a;
}

for $i ( 1 .. 10 ) {
    push @many, makeone();
}

print $many[4][5], "\n";

print "@many\n";
```

How can I free an array or hash so my program shrinks?

You usually can't. On most operating systems, memory allocated to a program can never be returned to the system. That's why long-running programs sometimes re-exec themselves. Some operating systems (notably, systems that use mmap(2) for allocating large chunks of memory) can reclaim memory that is no longer used, but on such systems, perl must be configured and compiled to use the OS's malloc, not perl's.

However, judicious use of my() on your variables will help make sure that they go out of scope so that Perl can free up that space for use in other parts of your program. A global variable, of course, never goes out of scope, so you can't get its space automatically reclaimed, although undef()ing and/or delete()ing it will achieve the same effect. In general, memory allocation and de-allocation isn't something you can or should be worrying about much in Perl, but even this capability (preallocation of data types) is in the works.

How can I make my CGI script more efficient?

Beyond the normal measures described to make general Perl programs faster or smaller, a CGI program has additional issues. It may be run several times per second. Given that each time it runs it will need to be re-compiled and will often allocate a megabyte or more of system memory, this can be a killer. Compiling into C **isn't going to help you** because the process start-up overhead is where the bottleneck is.

There are two popular ways to avoid this overhead. One solution involves running the Apache HTTP server (available from <http://www.apache.org/>) with either of the mod_perl or mod_fastcgi plugin modules.

With mod_perl and the Apache::Registry module (distributed with mod_perl), httpd will run with an embedded Perl interpreter which pre-compiles your script and then executes it within the same address space without forking. The Apache extension also gives Perl access to the internal server API, so modules written in Perl can do just about anything a module written in C can. For more on mod_perl, see <http://perl.apache.org/>

With the FCGI module (from CPAN) and the mod_fastcgi module (available from <http://www.fastcgi.com/>) each of your Perl programs becomes a permanent CGI daemon process.

Both of these solutions can have far-reaching effects on your system and on the way you write your CGI programs, so investigate them with care.

See http://www.cpan.org/modules/by-category/15_World_Wide_Web_HTML_HTTP_CGI/.

A non-free, commercial product, "The Velocity Engine for Perl", (<http://www.binevolve.com/> or

<http://www.binevolve.com/velocigen/>) might also be worth looking at. It will allow you to increase the performance of your Perl programs, running programs up to 25 times faster than normal CGI Perl when running in persistent Perl mode or 4 to 5 times faster without any modification to your existing CGI programs. Fully functional evaluation copies are available from the web site.

How can I hide the source for my Perl program?

Delete it. :-) Seriously, there are a number of (mostly unsatisfactory) solutions with varying levels of “security”.

First of all, however, you *can't* take away read permission, because the source code has to be readable in order to be compiled and interpreted. (That doesn't mean that a CGI script's source is readable by people on the web, though—only by people with access to the filesystem.) So you have to leave the permissions at the socially friendly 0755 level.

Some people regard this as a security problem. If your program does insecure things and relies on people not knowing how to exploit those insecurities, it is not secure. It is often possible for someone to determine the insecure things and exploit them without viewing the source. Security through obscurity, the name for hiding your bugs instead of fixing them, is little security indeed.

You can try using encryption via source filters (Starting from Perl 5.8 the `Filter::Simple` and `Filter::Util::Call` modules are included in the standard distribution), but any decent programmer will be able to decrypt it. You can try using the byte code compiler and interpreter described below, but the curious might still be able to de-compile it. You can try using the native-code compiler described below, but crackers might be able to disassemble it. These pose varying degrees of difficulty to people wanting to get at your code, but none can definitively conceal it (true of every language, not just Perl).

If you're concerned about people profiting from your code, then the bottom line is that nothing but a restrictive license will give you legal security. License your software and pepper it with threatening statements like “This is unpublished proprietary software of XYZ Corp. Your access to it does not give you permission to use it blah blah blah.” We are not lawyers, of course, so you should see a lawyer if you want to be sure your license's wording will stand up in court.

How can I compile my Perl program into byte code or C?

Malcolm Beattie has written a multifunction backend compiler, available from CPAN, that can do both these things. It is included in the perl5.005 release, but is still considered experimental. This means it's fun to play with if you're a programmer but not really for people looking for turn-key solutions.

Merely compiling into C does not in and of itself guarantee that your code will run very much faster. That's because except for lucky cases where a lot of native type inferencing is possible, the normal Perl run-time system is still present and so your program will take just as long to run and be just as big. Most programs save little more than compilation time, leaving execution no more than 10–30% faster. A few rare programs actually benefit significantly (even running several times faster), but this takes some tweaking of your code.

You'll probably be astonished to learn that the current version of the compiler generates a compiled form of your script whose executable is just as big as the original perl executable, and then some. That's because as currently written, all programs are prepared for a full `eval()` statement. You can tremendously reduce this cost by building a shared `libperl.so` library and linking against that. See the **INSTALL** podfile in the Perl source distribution for details. If you link your main perl binary with this, it will make it minuscule. For example, on one author's system, `/usr/bin/perl` is only 11k in size!

In general, the compiler will do nothing to make a Perl program smaller, faster, more portable, or more secure. In fact, it can make your situation worse. The executable will be bigger, your VM system may take longer to load the whole thing, the binary is fragile and hard to fix, and compilation never stopped software piracy in the form of crackers, viruses, or bootleggers. The real advantage of the compiler is merely packaging, and once you see the size of what it makes (well, unless you use a shared `libperl.so`), you'll probably want a complete Perl install anyway.

How can I compile Perl into Java?

You can also integrate Java and Perl with the Perl Resource Kit from O'Reilly and Associates. See <http://www.oreilly.com/catalog/prkunix/>.

Perl 5.6 comes with Java Perl Lingo, or JPL. JPL, still in development, allows Perl code to be called from Java. See `jpl/README` in the Perl source tree.

How can I get #!perl to work on [MS-DOS,NT,...]?

For OS/2 just use

```
extproc perl -S -your_switches
```

as the first line in *.cmd file (-S due to a bug in cmd.exe's 'extproc' handling). For DOS one should first invent a corresponding batch file and codify it in `ALTERNATIVE_SHEBANG` (see the *INSTALL* file in the source distribution for more information).

The Win95/NT installation, when using the ActiveState port of Perl, will modify the Registry to associate the .pl extension with the perl interpreter. If you install another port, perhaps even building your own Win95/NT Perl from the standard sources by using a Windows port of gcc (e.g., with cygwin or mingw32), then you'll have to modify the Registry yourself. In addition to associating .pl with the interpreter, NT people can use: `SET PATHEXT=%PATHEXT%.PL` to let them run the program `install-linux.pl` merely by typing `install-linux`.

Macintosh Perl programs will have the appropriate Creator and Type, so that double-clicking them will invoke the Perl application.

IMPORTANT!: Whatever you do, PLEASE don't get frustrated, and just throw the perl interpreter into your cgi-bin directory, in order to get your programs working for a web server. This is an **EXTREMELY** big security risk. Take the time to figure out how to do it correctly.

Can I write useful Perl programs on the command line?

Yes. Read *perlrun* for more information. Some examples follow. (These assume standard Unix shell quoting rules.)

```
# sum first and last fields
perl -lane 'print $F[0] + $F[-1]' *

# identify text files
perl -le 'for(@ARGV) {print if -f && -T _}' *

# remove (most) comments from C program
perl -0777 -pe 's{/\*.*?\*/}{ }gs' foo.c

# make file a month younger than today, defeating reaper daemons
perl -e '$X=24*60*60; utime(time(),time() + 30 * $X,@ARGV)' *

# find first unused uid
perl -le '$i++ while getpwuid($i); print $i'

# display reasonable manpath
echo $PATH | perl -nl -072 -e '
    s![^/+] *$!man!&&-d&&!$s{$_}++&&push@m,$_;END{print"@m"}'
```

OK, the last one was actually an Obfuscated Perl Contest entry. :-)

Why don't Perl one-liners work on my DOS/Mac/VMS system?

The problem is usually that the command interpreters on those systems have rather different ideas about quoting than the Unix shells under which the one-liners were created. On some systems, you may have to change single-quotes to double ones, which you must *NOT* do on Unix or Plan9 systems. You might also have to change a single % to a %%.

For example:

```
# Unix
perl -e 'print "Hello world\n"'

# DOS, etc.
perl -e "print \"Hello world\\n\\n\""

# Mac
print "Hello world\n"
    (then Run "Myscript" or Shift-Command-R)

# MPW
perl -e 'print "Hello world\n"'

# VMS
perl -e "print \"Hello world\\n\""
```

The problem is that none of these examples are reliable: they depend on the command interpreter. Under Unix, the first two often work. Under DOS, it's entirely possible that neither works. If 4DOS was the command shell, you'd probably have better luck like this:

```
perl -e "print <Ctrl-x>\"Hello world\\n<Ctrl-x>\""
```

Under the Mac, it depends which environment you are using. The MacPerl shell, or MPW, is much like Unix shells in its support for several quoting variants, except that it makes free use of the Mac's non-ASCII characters as control characters.

Using `qq()`, `q()`, and `qx()`, instead of "double quotes", 'single quotes', and 'backticks', may make one-liners easier to write.

There is no general solution to all of this. It is a mess.

[Some of this answer was contributed by Kenneth Albanowski.]

Where can I learn about CGI or Web programming in Perl?

For modules, get the CGI or LWP modules from CPAN. For textbooks, see the two especially dedicated to web stuff in the question on books. For problems and questions related to the web, like "Why do I get 500 Errors" or "Why doesn't it run from the browser right when it runs fine on the command line", see the troubleshooting guides and references in [perlfaq9](#) or in the CGI MetaFAQ:

http://www.perl.org/CGI_MetaFAQ.html

Where can I learn about object-oriented Perl programming?

A good place to start is [perltoot](#), and you can use [perlobj](#), [perlboot](#), [perltoot](#), [perltooc](#), and [perlbob](#) for reference. (If you are using really old Perl, you may not have all of these, try <http://www.perldoc.com/>, but consider upgrading your perl.)

A good book on OO on Perl is the "Object-Oriented Perl" by Damian Conway from Manning Publications, <http://www.manning.com/Conway/index.html>

Where can I learn about linking C with Perl? [h2xs, xsubpp]

If you want to call C from Perl, start with [perlxsut](#), moving on to [perlxs](#), [xsubpp](#), and [perlguts](#). If you want to call Perl from C, then read [perlembd](#), [perlcall](#), and [perlguts](#). Don't forget that you can learn a lot from looking at how the authors of existing extension modules wrote their code and solved their problems.

I've read perlembd, perlguts, etc., but I can't embed perl in

my C program; what am I doing wrong?

Download the ExtUtils::Embed kit from CPAN and run 'make test'. If the tests pass, read the pods again and again and again. If they fail, see [perlbug](#) and send a bug report with the output of `make test TEST_VERBOSE=1` along with `perl -V`.

When I tried to run my script, I got this message. What does it mean?

A complete list of Perl's error messages and warnings with explanatory text can be found in [perldiag](#). You can also use the `splain` program (distributed with Perl) to explain the error messages:

```
perl program 2>diag.out
splain [-v] [-p] diag.out
```

or change your program to explain the messages for you:

```
use diagnostics;

or

use diagnostics -verbose;
```

What's MakeMaker?

This module (part of the standard Perl distribution) is designed to write a Makefile for an extension module from a `Makefile.PL`. For more information, see [ExtUtils::MakeMaker](#).

AUTHOR AND COPYRIGHT

Copyright (c) 1997–2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples here are in the public domain. You are permitted and encouraged to use this code and any derivatives thereof in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit to the FAQ would be courteous but is not required.

NAME

perlfaq4 – Data Manipulation (\$Revision: 1.25 \$, \$Date: 2002/05/30 07:04:25 \$)

DESCRIPTION

This section of the FAQ answers questions related to manipulating numbers, dates, strings, arrays, hashes, and miscellaneous data issues.

Data: Numbers**Why am I getting long decimals (eg, 19.9499999999999) instead of the numbers I should be getting (eg, 19.95)?**

The infinite set that a mathematician thinks of as the real numbers can only be approximated on a computer, since the computer only has a finite number of bits to store an infinite number of, um, numbers.

Internally, your computer represents floating-point numbers in binary. Floating-point numbers read in from a file or appearing as literals in your program are converted from their decimal floating-point representation (eg, 19.95) to an internal binary representation.

However, 19.95 can't be precisely represented as a binary floating-point number, just like 1/3 can't be exactly represented as a decimal floating-point number. The computer's binary representation of 19.95, therefore, isn't exactly 19.95.

When a floating-point number gets printed, the binary floating-point representation is converted back to decimal. These decimal numbers are displayed in either the format you specify with `printf()`, or the current output format for numbers. (See *`$# in perlvar`* if you use `print`. `$#` has a different default value in Perl5 than it did in Perl4. Changing `$#` yourself is deprecated.)

This affects **all** computer languages that represent decimal floating-point numbers in binary, not just Perl. Perl provides arbitrary-precision decimal numbers with the `Math::BigFloat` module (part of the standard Perl distribution), but mathematical operations are consequently slower.

If precision is important, such as when dealing with money, it's good to work with integers and then divide at the last possible moment. For example, work in pennies (1995) instead of dollars and cents (19.95) and divide by 100 at the end.

To get rid of the superfluous digits, just use a format (eg, `printf("%.2f", 19.95)`) to get the required precision. See *[Floating-point Arithmetic in perlop](#)*.

Why isn't my octal data interpreted correctly?

Perl only understands octal and hex numbers as such when they occur as literals in your program. Octal literals in perl must start with a leading "0" and hexadecimal literals must start with a leading "0x". If they are read in from somewhere and assigned, no automatic conversion takes place. You must explicitly use `oct()` or `hex()` if you want the values converted to decimal. `oct()` interprets both hex ("0x350") numbers and octal ones ("0350" or even without the leading "0", like "377"), while `hex()` only converts hexadecimal ones, with or without a leading "0x", like "0x255", "3A", "ff", or "deadbeef". The inverse mapping from decimal to octal can be done with either the "%o" or "%O" `sprintf()` formats. To get from decimal to hex try either the "%x" or the "%X" formats to `sprintf()`.

This problem shows up most often when people try using `chmod()`, `mkdir()`, `umask()`, or `sysopen()`, which by widespread tradition typically take permissions in octal.

```
chmod(644, $file); # WRONG
chmod(0644, $file); # right
```

Note the mistake in the first line was specifying the decimal literal 644, rather than the intended octal literal 0644. The problem can be seen with:

```
printf("%#o", 644); # prints 01204
```

Surely you had not intended `chmod(01204, $file)`; – did you? If you want to use numeric literals as arguments to `chmod()` et al. then please try to express them as octal constants, that is with a leading zero

and with the following digits restricted to the set 0..7.

Does Perl have a `round()` function? What about `ceil()` and `floor()`? Trig functions?

Remember that `int()` merely truncates toward 0. For rounding to a certain number of digits, `sprintf()` or `printf()` is usually the easiest route.

```
printf("%.3f", 3.1415926535);      # prints 3.142
```

The POSIX module (part of the standard Perl distribution) implements `ceil()`, `floor()`, and a number of other mathematical and trigonometric functions.

```
use POSIX;
$ceil    = ceil(3.5);              # 4
$floor   = floor(3.5);            # 3
```

In 5.000 to 5.003 perls, trigonometry was done in the `Math::Complex` module. With 5.004, the `Math::Trig` module (part of the standard Perl distribution) implements the trigonometric functions. Internally it uses the `Math::Complex` module and some functions can break out from the real axis into the complex plane, for example the inverse sine of 2.

Rounding in financial applications can have serious implications, and the rounding method used should be specified precisely. In these cases, it probably pays not to trust whichever system rounding is being used by Perl, but to instead implement the rounding function you need yourself.

To see why, notice how you'll still have an issue on half-way-point alternation:

```
for ($i = 0; $i < 1.01; $i += 0.05) { printf "%.1f ", $i }
0.0 0.1 0.1 0.2 0.2 0.2 0.3 0.3 0.4 0.4 0.5 0.5 0.6 0.7 0.7
0.8 0.8 0.9 0.9 1.0 1.0
```

Don't blame Perl. It's the same as in C. IEEE says we have to do this. Perl numbers whose absolute values are integers under $2^{*}31$ (on 32 bit machines) will work pretty much like mathematical integers. Other numbers are not guaranteed.

How do I convert between numeric representations?

As always with Perl there is more than one way to do it. Below are a few examples of approaches to making common conversions between number representations. This is intended to be representational rather than exhaustive.

Some of the examples below use the `Bit::Vector` module from CPAN. The reason you might choose `Bit::Vector` over the perl built in functions is that it works with numbers of ANY size, that it is optimized for speed on some operations, and for at least some programmers the notation might be familiar.

How do I convert hexadecimal into decimal

Using perl's built in conversion of 0x notation:

```
$int = 0xDEADBEEF;
$dec = sprintf("%d", $int);
```

Using the hex function:

```
$int = hex("DEADBEEF");
$dec = sprintf("%d", $int);
```

Using pack:

```
$int = unpack("N", pack("H8", substr("0" x 8 . "DEADBEEF", -8)));
$dec = sprintf("%d", $int);
```

Using the CPAN module `Bit::Vector`:

```
use Bit::Vector;
$vec = Bit::Vector->new_Hex(32, "DEADBEEF");
```

```
$dec = $vec->to_Dec();
```

How do I convert from decimal to hexadecimal

Using sprintf:

```
$hex = sprintf("%X", 3735928559);
```

Using unpack

```
$hex = unpack("H*", pack("N", 3735928559));
```

Using Bit::Vector

```
use Bit::Vector;
$vec = Bit::Vector->new_Dec(32, -559038737);
$hex = $vec->to_Hex();
```

And Bit::Vector supports odd bit counts:

```
use Bit::Vector;
$vec = Bit::Vector->new_Dec(33, 3735928559);
$vec->Resize(32); # suppress leading 0 if unwanted
$hex = $vec->to_Hex();
```

How do I convert from octal to decimal

Using Perl's built in conversion of numbers with leading zeros:

```
$int = 033653337357; # note the leading 0!
$dec = sprintf("%d", $int);
```

Using the oct function:

```
$int = oct("33653337357");
$dec = sprintf("%d", $int);
```

Using Bit::Vector:

```
use Bit::Vector;
$vec = Bit::Vector->new(32);
$vec->Chunk_List_Store(3, split("//", reverse "33653337357"));
$dec = $vec->to_Dec();
```

How do I convert from decimal to octal

Using sprintf:

```
$oct = sprintf("%o", 3735928559);
```

Using Bit::Vector

```
use Bit::Vector;
$vec = Bit::Vector->new_Dec(32, -559038737);
$oct = reverse join('', $vec->Chunk_List_Read(3));
```

How do I convert from binary to decimal

Perl 5.6 lets you write binary numbers directly with the 0b notation:

```
$number = 0b10110110;
```

Using pack and ord

```
$decimal = ord(pack('B8', '10110110'));
```

Using pack and unpack for larger strings

```
$int = unpack("N", pack("B32",
    substr("0" x 32 . "11110101011011011111011101111", -32)));
```



```
$dec = sprintf("%d", $int);
```

```
# substr() is used to left pad a 32 character string with zeros.
```

Using Bit::Vector:

```
$vec = Bit::Vector->new_Bin(32, "11011110101011011011111011101111");
$dec = $vec->to_Dec();
```

How do I convert from decimal to binary

Using unpack;

```
$bin = unpack("B*", pack("N", 3735928559));
```

Using Bit::Vector:

```
use Bit::Vector;
$vec = Bit::Vector->new_Dec(32, -559038737);
$bin = $vec->to_Bin();
```

The remaining transformations (e.g. hex – oct, bin – hex, etc.) are left as an exercise to the inclined reader.

Why doesn't & work the way I want it to?

The behavior of binary arithmetic operators depends on whether they're used on numbers or strings. The operators treat a string as a series of bits and work with that (the string "3" is the bit pattern 00110011). The operators work with the binary form of a number (the number 3 is treated as the bit pattern 00000011).

So, saying `11 & 3` performs the "and" operation on numbers (yielding 1). Saying `"11" & "3"` performs the "and" operation on strings (yielding "1").

Most problems with `&` and `|` arise because the programmer thinks they have a number but really it's a string. The rest arise because the programmer says:

```
if ("\020\020" & "\101\101") {
    # ...
}
```

but a string consisting of two null bytes (the result of `"\020\020" & "\101\101"`) is not a false value in Perl. You need:

```
if ( ("\020\020" & "\101\101") !~ /^[\000]/ ) {
    # ...
}
```

How do I multiply matrices?

Use the `Math::Matrix` or `Math::MatrixReal` modules (available from CPAN) or the PDL extension (also available from CPAN).

How do I perform an operation on a series of integers?

To call a function on each element in an array, and collect the results, use:

```
@results = map { my_func($_) } @array;
```

For example:

```
@triple = map { 3 * $_ } @single;
```

To call a function on each element of an array, but ignore the results:

```
foreach $iterator (@array) {
    some_func($iterator);
}
```

To call a function on each integer in a (small) range, you **can** use:

```
@results = map { some_func($_) } (5 .. 25);
```

but you should be aware that the `..` operator creates an array of all integers in the range. This can take a lot of memory for large ranges. Instead use:

```
@results = ();
for ($i=5; $i < 500_005; $i++) {
    push(@results, some_func($i));
}
```

This situation has been fixed in Perl5.005. Use of `..` in a `for` loop will iterate over the range, without creating the entire range.

```
for my $i (5 .. 500_005) {
    push(@results, some_func($i));
}
```

will not create a list of 500,000 integers.

How can I output Roman numerals?

Get the <http://www.cpan.org/modules/by-module/Roman> module.

Why aren't my random numbers random?

If you're using a version of Perl before 5.004, you must call `srand` once at the start of your program to seed the random number generator. 5.004 and later automatically call `srand` at the beginning. Don't call `srand` more than once—you make your numbers less random, rather than more.

Computers are good at being predictable and bad at being random (despite appearances caused by bugs in your programs :-). see the *random* article in the "Far More Than You Ever Wanted To Know" collection in <http://www.cpan.org/olddoc/FMTEYEWTK.tgz>, courtesy of Tom Phoenix, talks more about this. John von Neumann said, "Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin."

If you want numbers that are more random than `rand` with `srand` provides, you should also check out the `Math::TrulyRandom` module from CPAN. It uses the imperfections in your system's timer to generate random numbers, but this takes quite a while. If you want a better pseudorandom generator than comes with your operating system, look at "Numerical Recipes in C" at <http://www.nr.com/>.

How do I get a random number between X and Y?

Use the following simple function. It selects a random integer between (and possibly including!) the two given integers, e.g., `random_int_in(50,120)`

```
sub random_int_in ($$) {
    my($min, $max) = @_;
    # Assumes that the two arguments are integers themselves!
    return $min if $min == $max;
    ($min, $max) = ($max, $min) if $min > $max;
    return $min + int rand(1 + $max - $min);
}
```

Data: Dates

How do I find the week-of-the-year/day-of-the-year?

The day of the year is in the array returned by `localtime()` (see *localtime in perlfunc*):

```
$day_of_year = (localtime(time()))[7];
```

How do I find the current century or millennium?

Use the following simple functions:

```
sub get_century {
    return int((((localtime(shift || time))[5] + 1999))/100);
}
sub get_millennium {
    return 1+int((((localtime(shift || time))[5] + 1899))/1000);
}
```

On some systems, you'll find that the POSIX module's `strftime()` function has been extended in a non-standard way to use a `%C` format, which they sometimes claim is the "century". It isn't, because on most such systems, this is only the first two digits of the four-digit year, and thus cannot be used to reliably determine the current century or millennium.

How can I compare two dates and find the difference?

If you're storing your dates as epoch seconds then simply subtract one from the other. If you've got a structured date (distinct year, day, month, hour, minute, seconds values), then for reasons of accessibility, simplicity, and efficiency, merely use either `timelocal` or `timegm` (from the `Time::Local` module in the standard distribution) to reduce structured dates to epoch seconds. However, if you don't know the precise format of your dates, then you should probably use either of the `Date::Manip` and `Date::Calc` modules from CPAN before you go hacking up your own parsing routine to handle arbitrary date formats.

How can I take a string and turn it into epoch seconds?

If it's a regular enough string that it always has the same format, you can split it up and pass the parts to `timelocal` in the standard `Time::Local` module. Otherwise, you should look into the `Date::Calc` and `Date::Manip` modules from CPAN.

How can I find the Julian Day?

Use the `Time::JulianDay` module (part of the `Time`-modules bundle available from CPAN.)

Before you immerse yourself too deeply in this, be sure to verify that it is the *Julian* Day you really want. Are you interested in a way of getting serial days so that you just can tell how many days they are apart or so that you can do also other date arithmetic? If you are interested in performing date arithmetic, this can be done using modules `Date::Manip` or `Date::Calc`.

There is too many details and much confusion on this issue to cover in this FAQ, but the term is applied (correctly) to a calendar now supplanted by the Gregorian Calendar, with the Julian Calendar failing to adjust properly for leap years on centennial years (among other annoyances). The term is also used (incorrectly) to mean: [1] days in the Gregorian Calendar; and [2] days since a particular starting time or 'epoch', usually 1970 in the Unix world and 1980 in the MS-DOS/Windows world. If you find that it is not the first meaning that you really want, then check out the `Date::Manip` and `Date::Calc` modules. (Thanks to David Cassell for most of this text.)

How do I find yesterday's date?

The `time()` function returns the current time in seconds since the epoch. Take twenty-four hours off that:

```
$yesterday = time() - ( 24 * 60 * 60 );
```

Then you can pass this to `localtime()` and get the individual year, month, day, hour, minute, seconds values.

Note very carefully that the code above assumes that your days are twenty-four hours each. For most people, there are two days a year when they aren't: the switch to and from summer time throws this off. A solution to this issue is offered by Russ Allbery.

```
sub yesterday {
    my $now = defined $_[0] ? $_[0] : time;
    my $then = $now - 60 * 60 * 24;
```

```

    my $ndst = (localtime $now)[8] > 0;
    my $dst = (localtime $then)[8] > 0;
    $then - ($dst - $ndst) * 60 * 60;
}
# Should give you "this time yesterday" in seconds since epoch relative to
# the first argument or the current time if no argument is given and
# suitable for passing to localtime or whatever else you need to do with
# it. $ndst is whether we're currently in daylight savings time; $dst is
# whether the point 24 hours ago was in daylight savings time. If $dst
# and $ndst are the same, a boundary wasn't crossed, and the correction
# will subtract 0. If $dst is 1 and $ndst is 0, subtract an hour more
# from yesterday's time since we gained an extra hour while going off
# daylight savings time. If $dst is 0 and $ndst is 1, subtract a
# negative hour (add an hour) to yesterday's time since we lost an hour.
#
# All of this is because during those days when one switches off or onto
# DST, a "day" isn't 24 hours long; it's either 23 or 25.
#
# The explicit settings of $ndst and $dst are necessary because localtime
# only says it returns the system tm struct, and the system tm struct at
# least on Solaris doesn't guarantee any particular positive value (like,
# say, 1) for isdst, just a positive value. And that value can
# potentially be negative, if DST information isn't available (this sub
# just treats those cases like no DST).
#
# Note that between 2am and 3am on the day after the time zone switches
# off daylight savings time, the exact hour of "yesterday" corresponding
# to the current hour is not clearly defined. Note also that if used
# between 2am and 3am the day after the change to daylight savings time,
# the result will be between 3am and 4am of the previous day; it's
# arguable whether this is correct.
#
# This sub does not attempt to deal with leap seconds (most things don't).
#
# Copyright relinquished 1999 by Russ Allbery <rra@stanford.edu>
# This code is in the public domain

```

Does Perl have a Year 2000 problem? Is Perl Y2K compliant?

Short answer: No, Perl does not have a Year 2000 problem. Yes, Perl is Y2K compliant (whatever that means). The programmers you've hired to use it, however, probably are not.

Long answer: The question belies a true understanding of the issue. Perl is just as Y2K compliant as your pencil—no more, and no less. Can you use your pencil to write a non-Y2K-compliant memo? Of course you can. Is that the pencil's fault? Of course it isn't.

The date and time functions supplied with Perl (gmtime and localtime) supply adequate information to determine the year well beyond 2000 (2038 is when trouble strikes for 32-bit machines). The year returned by these functions when used in a list context is the year minus 1900. For years between 1910 and 1999 this *happens* to be a 2-digit decimal number. To avoid the year 2000 problem simply do not treat the year as a 2-digit number. It isn't.

When gmtime() and localtime() are used in scalar context they return a timestamp string that contains a fully-expanded year. For example, \$timestamp = gmtime(1005613200) sets \$timestamp to "Tue Nov 13 01:00:00 2001". There's no year 2000 problem here.

That doesn't mean that Perl can't be used to create non-Y2K compliant programs. It can. But so can your pencil. It's the fault of the user, not the language. At the risk of inflaming the NRA: "Perl doesn't break

Y2K, people do.” See <http://language.perl.com/news/y2k.html> for a longer exposition.

Data: Strings

How do I validate input?

The answer to this question is usually a regular expression, perhaps with auxiliary logic. See the more specific questions (numbers, mail addresses, etc.) for details.

How do I unescape a string?

It depends just what you mean by “escape”. URL escapes are dealt with in [perlfaq9](#). Shell escapes with the backslash (\) character are removed with

```
s/\\(\\|\\.|\\n|\\t|\\f|\\r)/$1/g;
```

This won’t expand “\n” or “\t” or any other special escapes.

How do I remove consecutive pairs of characters?

To turn “abbcccd” into “abccd”:

```
s/(.)\\1/$1/g;          # add /s to include newlines
```

Here’s a solution that turns “abbcccd” to “abcd”:

```
y///cs;          # y == tr, but shorter :-)
```

How do I expand function calls in a string?

This is documented in [perlref](#). In general, this is fraught with quoting and readability problems, but it is possible. To interpolate a subroutine call (in list context) into a string:

```
print "My sub returned @{[mysub(1,2,3)]} that time.\n";
```

If you prefer scalar context, similar chicanery is also useful for arbitrary expressions:

```
print "That yields ${\\($n + 5)} widgets\n";
```

Version 5.004 of Perl had a bug that gave list context to the expression in `${...}`, but this is fixed in version 5.005.

See also “How can I expand variables in text strings?” in this section of the FAQ.

How do I find matching/nesting anything?

This isn’t something that can be done in one regular expression, no matter how complicated. To find something between two single characters, a pattern like `/x([^x]*)x/` will get the intervening bits in `$1`. For multiple ones, then something more like `/alpha(.*)omega/` would be needed. But none of these deals with nested patterns, nor can they. For that you’ll have to write a parser.

If you are serious about writing a parser, there are a number of modules or oddities that will make your life a lot easier. There are the CPAN modules `Parse::RecDescent`, `Parse::Yapp`, and `Text::Balanced`; and the yacc program. Starting from perl 5.8 the `Text::Balanced` is part of the standard distribution.

One simple destructive, inside–out approach that you might try is to pull out the smallest nesting parts one at a time:

```
while (s/BEGIN(?:(!BEGIN) (!END) .)*END//gs) {
    # do something with $1
}
```

A more complicated and sneaky approach is to make Perl’s regular expression engine do it for you. This is courtesy Dean Inada, and rather has the nature of an Obfuscated Perl Contest entry, but it really does work:

```
# $_ contains the string to parse
# BEGIN and END are the opening and closing markers for the
# nested text.
```

```
@( = ( ' ( ' , ' ' ) ;
@) = ( ' ) ' , ' ' ) ;
($re=$_) =~ s/((BEGIN) | (END) | .)/$) [!$3] \Q$1\E$([!$2])/gs;
@$ = (eval{/$re/},$@!~/unmatched/i);
print join("\n",@$[0..$#$]) if( $$[-1] );
```

How do I reverse a string?

Use `reverse()` in scalar context, as documented in [reverse](#).

```
$reversed = reverse $string;
```

How do I expand tabs in a string?

You can do it yourself:

```
1 while $string =~ s/\t+/' ' x (length($&) * 8 - length($`) % 8)/e;
```

Or you can just use the `Text::Tabs` module (part of the standard Perl distribution).

```
use Text::Tabs;
@expanded_lines = expand(@lines_with_tabs);
```

How do I reformat a paragraph?

Use `Text::Wrap` (part of the standard Perl distribution):

```
use Text::Wrap;
print wrap("\t", ' ', @paragraphs);
```

The paragraphs you give to `Text::Wrap` should not contain embedded newlines. `Text::Wrap` doesn't justify the lines (flush-right).

Or use the CPAN module `Text::Autoformat`. Formatting files can be easily done by making a shell alias, like so:

```
alias fmt="perl -i -MText::Autoformat -n0777 \
-e 'print autoformat $_, {all=>1}' $*"
```

See the documentation for `Text::Autoformat` to appreciate its many capabilities.

How can I access/change the first N letters of a string?

There are many ways. If you just want to grab a copy, use `substr()`:

```
$first_byte = substr($a, 0, 1);
```

If you want to modify part of a string, the simplest way is often to use `substr()` as an lvalue:

```
substr($a, 0, 3) = "Tom";
```

Although those with a pattern matching kind of thought process will likely prefer

```
$a =~ s/^\.../Tom/;
```

How do I change the Nth occurrence of something?

You have to keep track of N yourself. For example, let's say you want to change the fifth occurrence of "whoever" or "whomever" into "whosoever" or "whomsoever", case insensitively. These all assume that `$_` contains the string to be altered.

```
$count = 0;
s{((whom?)ever)}{
    ++$count == 5          # is it the 5th?
    ? "${2}soever"        # yes, swap
    : $1                  # renege and leave it there
}ige;
```

In the more general case, you can use the `/g` modifier in a while loop, keeping count of matches.

```

$WANT = 3;
$count = 0;
$_ = "One fish two fish red fish blue fish";
while (/(\\w+)\\s+fish\\b/gi) {
    if (++$count == $WANT) {
        print "The third fish is a $1 one.\\n";
    }
}

```

That prints out: "The third fish is a red one." You can also use a repetition count and repeated pattern like this:

```
/(?:\\w+\\s+fish\\s+){2}(\\w+)\\s+fish/i;
```

How can I count the number of occurrences of a substring within a string?

There are a number of ways, with varying efficiency. If you want a count of a certain single character (X) within a string, you can use the `tr///` function like so:

```

$string = "ThisXlineXhasXsomeXx'sXinXit";
$count = ($string =~ tr/X//);
print "There are $count X characters in the string";

```

This is fine if you are just looking for a single character. However, if you are trying to count multiple character substrings within a larger string, `tr///` won't work. What you can do is wrap a `while()` loop around a global pattern match. For example, let's count negative integers:

```

$string = "-9 55 48 -2 23 -76 4 14 -44";
while ($string =~ /-\\d+/g) { $count++ }
print "There are $count negative numbers in the string";

```

Another version uses a global match in list context, then assigns the result to a scalar, producing a count of the number of matches.

```
$count = () = $string =~ /-\\d+/g;
```

How do I capitalize all the words on one line?

To make the first letter of each word upper case:

```
$line =~ s/\\b(\\w)/\\U$1/g;
```

This has the strange effect of turning "don't do it" into "Don'T Do It". Sometimes you might want this. Other times you might need a more thorough solution (Suggested by brian d foy):

```

$string =~ s/ (
    (\\^\\w)      #at the beginning of the line
    |          # or
    (\\s\\w)     #preceded by whitespace
)
    /\\U$1/xg;
$string =~ /(\\[\\w']+)/\\u\\L$1/g;

```

To make the whole line upper case:

```
$line = uc($line);
```

To force each word to be lower case, with the first letter upper case:

```
$line =~ s/(\\w+)/\\u\\L$1/g;
```

You can (and probably should) enable locale awareness of those characters by placing a `use locale` pragma in your program. See [perllocale](#) for endless details on locales.

This is sometimes referred to as putting something into "title case", but that's not quite accurate. Consider

the proper capitalization of the movie *Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb*, for example.

How can I split a [character] delimited string except when inside

[character]? (Comma-separated files)

Take the example case of trying to split a string that is comma-separated into its different fields. (We'll pretend you said comma-separated, not comma-delimited, which is different and almost never what you mean.) You can't use `split(/,/)` because you shouldn't split if the comma is inside quotes. For example, take a data line like this:

```
SAR001,"","Cimetrix, Inc","Bob Smith","CAM",N,8,1,0,7,"Error, Core Dumped"
```

Due to the restriction of the quotes, this is a fairly complex problem. Thankfully, we have Jeffrey Friedl, author of a highly recommended book on regular expressions, to handle these for us. He suggests (assuming your string is contained in `$text`) :

```
@new = ();
push(@new, $+) while $text =~ m{
    "([^\\"\\\\]*(?:\\.["\\\\]*))*",? # groups the phrase inside the quotes
    | ([^,]+),?
    | ,
}gx;
push(@new, undef) if substr($text,-1,1) eq ',';
```

If you want to represent quotation marks inside a quotation-mark-delimited field, escape them with backslashes (eg, "like \"this\""). Unescaping them is a task addressed earlier in this section.

Alternatively, the `Text::ParseWords` module (part of the standard Perl distribution) lets you say:

```
use Text::ParseWords;
@new = quotewords(",", 0, $text);
```

There's also a `Text::CSV` (Comma-Separated Values) module on CPAN.

How do I strip blank space from the beginning/end of a string?

Although the simplest approach would seem to be

```
$string =~ s/^\s*(.*?)\s*$/\1/;
```

not only is this unnecessarily slow and destructive, it also fails with embedded newlines. It is much faster to do this operation in two steps:

```
$string =~ s/^\s+//;
$string =~ s/\s+$//;
```

Or more nicely written as:

```
for ($string) {
    s/^\s+//;
    s/\s+$//;
}
```

This idiom takes advantage of the `foreach` loop's aliasing behavior to factor out common code. You can do this on several strings at once, or arrays, or even the values of a hash if you use a slice:

```
# trim whitespace in the scalar, the array,
# and all the values in the hash
foreach ($scalar, @array, @hash{keys %hash}) {
    s/^\s+//;
    s/\s+$//;
}
```


How do I pad a string with blanks or pad a number with zeroes?

(This answer contributed by Uri Guttman, with kibitzing from Bart Lateur.)

In the following examples, `$pad_len` is the length to which you wish to pad the string, `$text` or `$num` contains the string to be padded, and `$pad_char` contains the padding character. You can use a single character string constant instead of the `$pad_char` variable if you know what it is in advance. And in the same way you can use an integer in place of `$pad_len` if you know the pad length in advance.

The simplest method uses the `sprintf` function. It can pad on the left or right with blanks and on the left with zeroes and it will not truncate the result. The `pack` function can only pad strings on the right with blanks and it will truncate the result to a maximum length of `$pad_len`.

```
# Left padding a string with blanks (no truncation):
$padding = sprintf("%${pad_len}s", $text);

# Right padding a string with blanks (no truncation):
$padding = sprintf("%-${pad_len}s", $text);

# Left padding a number with 0 (no truncation):
$padding = sprintf("%0${pad_len}d", $num);

# Right padding a string with blanks using pack (will truncate):
$padding = pack("A${pad_len}", $text);
```

If you need to pad with a character other than blank or zero you can use one of the following methods. They all generate a pad string with the `x` operator and combine that with `$text`. These methods do not truncate `$text`.

Left and right padding with any character, creating a new string:

```
$padding = $pad_char x ( $pad_len - length( $text ) ) . $text;
$padding = $text . $pad_char x ( $pad_len - length( $text ) );
```

Left and right padding with any character, modifying `$text` directly:

```
substr( $text, 0, 0 ) = $pad_char x ( $pad_len - length( $text ) );
$text .= $pad_char x ( $pad_len - length( $text ) );
```

How do I extract selected columns from a string?

Use `substr()` or `unpack()`, both documented in [perlfunc](#). If you prefer thinking in terms of columns instead of widths, you can use this kind of thing:

```
# determine the unpack format needed to split Linux ps output
# arguments are cut columns
my $fmt = cut2fmt(8, 14, 20, 26, 30, 34, 41, 47, 59, 63, 67, 72);

sub cut2fmt {
    my(@positions) = @_;
    my $template = '';
    my $lastpos = 1;
    for my $place (@positions) {
        $template .= "A" . ($place - $lastpos) . " ";
        $lastpos = $place;
    }
    $template .= "A*";
    return $template;
}
```

How do I find the soundex value of a string?

Use the standard `Text::Soundex` module distributed with Perl. Before you do so, you may want to determine whether 'soundex' is in fact what you think it is. Knuth's soundex algorithm compresses words into a small

space, and so it does not necessarily distinguish between two words which you might want to appear separately. For example, the last names ‘Knuth’ and ‘Kant’ are both mapped to the soundex code K530. If `Text::Soundex` does not do what you are looking for, you might want to consider the `String::Approx` module available at CPAN.

How can I expand variables in text strings?

Let’s assume that you have a string like:

```
$text = 'this has a $foo in it and a $bar';
```

If those were both global variables, then this would suffice:

```
$text =~ s/\$(\w+)/${$1}/g; # no /e needed
```

But since they are probably lexicals, or at least, they could be, you’d have to do this:

```
$text =~ s/(\$(\w+))/$1/eeg;
die if $@; # needed /ee, not /e
```

It’s probably better in the general case to treat those variables as entries in some special hash. For example:

```
%user_defs = (
    foo => 23,
    bar => 19,
);
$text =~ s/\$(\w+)/$user_defs{$1}/g;
```

See also “How do I expand function calls in a string?” in this section of the FAQ.

What’s wrong with always quoting "\$vars"?

The problem is that those double-quotes force stringification—coercing numbers and references into strings—even when you don’t want them to be strings. Think of it this way: double-quote expansion is used to produce new strings. If you already have a string, why do you need more?

If you get used to writing odd things like these:

```
print "$var"; # BAD
$new = "$old"; # BAD
somefunc("$var"); # BAD
```

You’ll be in trouble. Those should (in 99.8% of the cases) be the simpler and more direct:

```
print $var;
$new = $old;
somefunc($var);
```

Otherwise, besides slowing you down, you’re going to break code when the thing in the scalar is actually neither a string nor a number, but a reference:

```
func(\@array);
sub func {
    my $aref = shift;
    my $oref = "$aref"; # WRONG
}
```

You can also get into subtle problems on those few operations in Perl that actually do care about the difference between a string and a number, such as the magical `++` autoincrement operator or the `syscall()` function.

Stringification also destroys arrays.

```
@lines = `command`;
print "@lines"; # WRONG - extra blanks
print @lines; # right
```

Why don't my <<HERE documents work?

Check for these three things:

1. There must be no space after the << part.
2. There (probably) should be a semicolon at the end.
3. You can't (easily) have any space in front of the tag.

If you want to indent the text in the here document, you can do this:

```
# all in one
($VAR = <<HERE_TARGET) =~ s/^\s+//gm;
    your text
    goes here
HERE_TARGET
```

But the HERE_TARGET must still be flush against the margin. If you want that indented also, you'll have to quote in the indentation.

```
($quote = <<'    FINIS') =~ s/^\s+//gm;
    ...we will have peace, when you and all your works have
    perished--and the works of your dark master to whom you
    would deliver us. You are a liar, Saruman, and a corrupter
    of men's hearts. --Theoden in /usr/src/perl/taint.c
    FINIS
$quote =~ s/\s+--/\n--/;
```

A nice general-purpose fixer-upper function for indented here documents follows. It expects to be called with a here document as its argument. It looks to see whether each line begins with a common substring, and if so, strips that substring off. Otherwise, it takes the amount of leading whitespace found on the first line and removes that much off each subsequent line.

```
sub fix {
    local $_ = shift;
    my ($white, $leader); # common whitespace and common leading string
    if (/^\s*(?:([^\w\s]+)(\s*).*)\n(?:\s*\1\2?.*\n)+$/ ) {
        ($white, $leader) = ($2, quotemeta($1));
    } else {
        ($white, $leader) = (/^\s+/ , '');
    }
    s/^\s*?$leader(?:$white)?//gm;
    return $_;
}
```

This works with leading special strings, dynamically determined:

```
$remember_the_main = fix<<'    MAIN_INTERPRETER_LOOP';
    @@@ int
    @@@ runops() {
    @@@     SAVEI32(runlevel);
    @@@     runlevel++;
    @@@     while ( op = (*op->op_ppaddr)() );
    @@@     TAINNT_NOT;
    @@@     return 0;
    @@@ }
MAIN_INTERPRETER_LOOP
```

Or with a fixed amount of leading whitespace, with remaining indentation correctly preserved:

```
$poem = fix<<EVER_ON_AND_ON;
```

```

Now far ahead the Road has gone,
    And I must follow, if I can,
Pursuing it with eager feet,
    Until it joins some larger way
Where many paths and errands meet.
    And whither then? I cannot say.
        --Bilbo in /usr/src/perl/pp_ctl.c
EVER_ON_AND_ON

```

Data: Arrays

What is the difference between a list and an array?

An array has a changeable length. A list does not. An array is something you can push or pop, while a list is a set of values. Some people make the distinction that a list is a value while an array is a variable. Subroutines are passed and return lists, you put things into list context, you initialize arrays with lists, and you `foreach()` across a list. `@` variables are arrays, anonymous arrays are arrays, arrays in scalar context behave like the number of elements in them, subroutines access their arguments through the array `@_`, and `push/pop/shift` only work on arrays.

As a side note, there's no such thing as a list in scalar context. When you say

```
$scalar = (2, 5, 7, 9);
```

you're using the comma operator in scalar context, so it uses the scalar comma operator. There never was a list there at all! This causes the last value to be returned: 9.

What is the difference between `$array[1]` and `@array[1]`?

The former is a scalar value; the latter an array slice, making it a list with one (scalar) value. You should use `$` when you want a scalar value (most of the time) and `@` when you want a list with one scalar value in it (very, very rarely; nearly never, in fact).

Sometimes it doesn't make a difference, but sometimes it does. For example, compare:

```
$good[0] = 'some program that outputs several lines';
```

with

```
@bad[0] = 'same program that outputs several lines';
```

The `use warnings` pragma and the `-w` flag will warn you about these matters.

How can I remove duplicate elements from a list or array?

There are several possible ways, depending on whether the array is ordered and whether you wish to preserve the ordering.

- a) If `@in` is sorted, and you want `@out` to be sorted: (this assumes all true values in the array)

```

$prev = "not equal to $in[0]";
@out = grep($_ ne $prev && ($prev = $_, 1), @in);

```

This is nice in that it doesn't use much extra memory, simulating `uniq(1)`'s behavior of removing only adjacent duplicates. The `", 1"` guarantees that the expression is true (so that `grep` picks it up) even if the `$_` is 0, "", or `undef`.

- b) If you don't know whether `@in` is sorted:

```

undef %saw;
@out = grep(!$saw{$_}++, @in);

```

- c) Like (b), but `@in` contains only small integers:

```
@out = grep(!$saw[$_]++, @in);
```

- d) A way to do (b) without any loops or greps:

```
undef %saw;
@saw{@in} = ();
@out = sort keys %saw; # remove sort if undesired
```

- e) Like (d), but @in contains only small positive integers:

```
undef @ary;
@ary{@in} = @in;
@out = grep {defined} @ary;
```

But perhaps you should have been using a hash all along, eh?

How can I tell whether a certain element is contained in a list or array?

Hearing the word "in" is an *indication* that you probably should have used a hash, not a list or array, to store your data. Hashes are designed to answer this question quickly and efficiently. Arrays aren't.

That being said, there are several ways to approach this. If you are going to make this query many times over arbitrary string values, the fastest way is probably to invert the original array and maintain a hash whose keys are the first array's values.

```
@blues = qw/azure cerulean teal turquoise lapis-lazuli/;
%is_blue = ();
for (@blues) { $is_blue{$_} = 1 }
```

Now you can check whether `$is_blue{$some_color}`. It might have been a good idea to keep the blues all in a hash in the first place.

If the values are all small integers, you could use a simple indexed array. This kind of an array will take up less space:

```
@primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);
@is_tiny_prime = ();
for (@primes) { $is_tiny_prime[$_] = 1 }
# or simply @istiny_prime[@primes] = (1) x @primes;
```

Now you check whether `$is_tiny_prime[$some_number]`.

If the values in question are integers instead of strings, you can save quite a lot of space by using bit strings instead:

```
@articles = ( 1..10, 150..2000, 2017 );
undef $read;
for (@articles) { vec($read,$_ ,1) = 1 }
```

Now check whether `vec($read,$n,1)` is true for some `$n`.

Please do not use

```
($is_there) = grep $_ eq $whatever, @array;
```

or worse yet

```
($is_there) = grep /$whatever/, @array;
```

These are slow (checks every element even if the first matches), inefficient (same reason), and potentially buggy (what if there are regex characters in `$whatever`?) . If you're only testing once, then use:

```
$is_there = 0;
foreach $elt (@array) {
    if ($elt eq $elt_to_find) {
        $is_there = 1;
        last;
    }
}
```

```

    }
}
if ($is_there) { ... }

```

How do I compute the difference of two arrays? How do I compute the intersection of two arrays?

Use a hash. Here's code to do both and more. It assumes that each element is unique in a given array:

```

@union = @intersection = @difference = ();
%count = ();
foreach $element (@array1, @array2) { $count{$element}++ }
foreach $element (keys %count) {
    push @union, $element;
    push @{ $count{$element} > 1 ? \@intersection : \@difference }, $element;
}

```

Note that this is the *symmetric difference*, that is, all elements in either A or in B but not in both. Think of it as an xor operation.

How do I test whether two arrays or hashes are equal?

The following code works for single-level arrays. It uses a stringwise comparison, and does not distinguish defined versus undefined empty strings. Modify if you have other needs.

```

$are_equal = compare_arrays(\@frogs, \@toads);

sub compare_arrays {
    my ($first, $second) = @_;
    no warnings; # silence spurious -w undef complaints
    return 0 unless @$first == @$second;
    for (my $i = 0; $i < @$first; $i++) {
        return 0 if $first->[$i] ne $second->[$i];
    }
    return 1;
}

```

For multilevel structures, you may wish to use an approach more like this one. It uses the CPAN module FreezeThaw:

```

use FreezeThaw qw(cmpStr);
@a = @b = ( "this", "that", [ "more", "stuff" ] );

printf "a and b contain %s arrays\n",
    cmpStr(\@a, \@b) == 0
    ? "the same"
    : "different";

```

This approach also works for comparing hashes. Here we'll demonstrate two different answers:

```

use FreezeThaw qw(cmpStr cmpStrHard);

%a = %b = ( "this" => "that", "extra" => [ "more", "stuff" ] );
$a{EXTRA} = \%b;
$b{EXTRA} = \%a;

printf "a and b contain %s hashes\n",
    cmpStr(\%a, \%b) == 0 ? "the same" : "different";

printf "a and b contain %s hashes\n",
    cmpStrHard(\%a, \%b) == 0 ? "the same" : "different";

```

The first reports that both those the hashes contain the same data, while the second reports that they do not. Which you prefer is left as an exercise to the reader.

How do I find the first array element for which a condition is true?

You can use this if you care about the index:

```
for ($i= 0; $i < @array; $i++) {
    if ($array[$i] eq "Waldo") {
        $found_index = $i;
        last;
    }
}
```

Now `$found_index` has what you want.

How do I handle linked lists?

In general, you usually don't need a linked list in Perl, since with regular arrays, you can push and pop or shift and unshift at either end, or you can use splice to add and/or remove arbitrary number of elements at arbitrary points. Both pop and shift are both O(1) operations on Perl's dynamic arrays. In the absence of shifts and pops, push in general needs to reallocate on the order every log(N) times, and unshift will need to copy pointers each time.

If you really, really wanted, you could use structures as described in [perldsc](#) or [perltoot](#) and do just what the algorithm book tells you to do. For example, imagine a list node like this:

```
$node = {
    VALUE => 42,
    LINK  => undef,
};
```

You could walk the list this way:

```
print "List: ";
for ($node = $head; $node; $node = $node->{LINK}) {
    print $node->{VALUE}, " ";
}
print "\n";
```

You could add to the list this way:

```
my ($head, $tail);
$tail = append($head, 1);          # grow a new head
for $value ( 2 .. 10 ) {
    $tail = append($tail, $value);
}

sub append {
    my($list, $value) = @_;
    my $node = { VALUE => $value };
    if ($list) {
        $node->{LINK} = $list->{LINK};
        $list->{LINK} = $node;
    } else {
        $_[0] = $node;          # replace caller's version
    }
    return $node;
}
```

But again, Perl's built-in are virtually always good enough.

How do I handle circular lists?

Circular lists could be handled in the traditional fashion with linked lists, or you could just do something like this with an array:

```
unshift(@array, pop(@array)); # the last shall be first
push(@array, shift(@array)); # and vice versa
```

How do I shuffle an array randomly?

If you either have Perl 5.8.0 or later installed, or if you have `Scalar-List-Utils` 1.03 or later installed, you can say:

```
use List::Util 'shuffle';
@shuffled = shuffle(@list);
```

If not, you can use a Fisher-Yates shuffle.

```
sub fisher_yates_shuffle {
    my $deck = shift; # $deck is a reference to an array
    my $i = @$deck;
    while ($i--) {
        my $j = int rand ($i+1);
        @$deck[$i,$j] = @$deck[$j,$i];
    }
}

# shuffle my mpeg collection
#
my @mpeg = <audio/*/*.mp3>;
fisher_yates_shuffle( \@mpeg ); # randomize @mpeg in place
print @mpeg;
```

Note that the above implementation shuffles an array in place, unlike the `List::Util::shuffle()` which takes a list and returns a new shuffled list.

You've probably seen shuffling algorithms that work using `splice`, randomly picking another element to swap the current element with

```
srand;
@new = ();
@old = 1 .. 10; # just a demo
while (@old) {
    push(@new, splice(@old, rand @old, 1));
}
```

This is bad because `splice` is already $O(N)$, and since you do it N times, you just invented a quadratic algorithm; that is, $O(N^2)$. This does not scale, although Perl is so efficient that you probably won't notice this until you have rather largish arrays.

How do I process/modify each element of an array?

Use `for/foreach`:

```
for (@lines) {
    s/foo/bar/; # change that word
    y/XZ/ZX/; # swap those letters
}
```

Here's another; let's compute spherical volumes:

```
for (@volumes = @radii) { # @volumes has changed parts
    $_ **= 3;
```



```

        $_ *= (4/3) * 3.14159; # this will be constant folded
    }

```

If you want to do the same thing to modify the values of the hash, you can use the `values` function. As of Perl 5.6 the values are not copied, so if you modify `$orbit` (in this case), you modify the value.

```

    for $orbit ( values %orbits ) {
        ($orbit **= 3) *= (4/3) * 3.14159;
    }

```

Prior to perl 5.6 `values` returned copies of the values, so older perl code often contains constructions such as `@orbits{keys %orbits}` instead of `values %orbits` where the hash is to be modified.

How do I select a random element from an array?

Use the `rand()` function (see [rand](#)):

```

# at the top of the program:
srand;                                # not needed for 5.004 and later

# then later on
$index = rand @array;
$element = $array[$index];

```

Make sure you *only call `srand` once per program, if then*. If you are calling it more than once (such as before each call to `rand`), you're almost certainly doing something wrong.

How do I permute N elements of a list?

Here's a little program that generates all permutations of all the words on each line of input. The algorithm embodied in the `permute()` function should work on any list:

```

#!/usr/bin/perl -n
# tsc-permute: permute each word of input
permute([split], []);
sub permute {
    my @items = @{ $_[0] };
    my @perms = @{ $_[1] };
    unless (@items) {
        print "@perms\n";
    } else {
        my(@newitems, @newperms, $i);
        foreach $i (0 .. $#items) {
            @newitems = @items;
            @newperms = @perms;
            unshift(@newperms, splice(@newitems, $i, 1));
            permute([@newitems], [@newperms]);
        }
    }
}

```

Unfortunately, this algorithm is very inefficient. The `Algorithm::Permute` module from CPAN runs at least an order of magnitude faster. If you don't have a C compiler (or a binary distribution of `Algorithm::Permute`), then you can use `List::Permutor` which is written in pure Perl, and is still several times faster than the algorithm above.

How do I sort an array by (anything)?

Supply a comparison function to `sort()` (described in [sort](#)):

```

@list = sort { $a <=> $b } @list;

```

The default sort function is `cmp`, string comparison, which would sort `(1, 2, 10)` into `(1, 10, 2)`. <

`<=` , used above, is the numerical comparison operator.

If you have a complicated function needed to pull out the part you want to sort on, then don't do it inside the sort function. Pull it out first, because the sort BLOCK can be called many times for the same element. Here's an example of how to pull out the first word after the first number on each item, and then sort those words case-insensitively.

```
@idx = ();
for (@data) {
    ($item) = /\d+\s*(\S+)/;
    push @idx, uc($item);
}
@sorted = @data[ sort { $idx[$a] cmp $idx[$b] } 0 .. $#idx ];
```

which could also be written this way, using a trick that's come to be known as the Schwartzian Transform:

```
@sorted = map { $_->[0] }
    sort { $a->[1] cmp $b->[1] }
    map { [ $_, uc( /\d+\s*(\S+)/ ) [0] ] } @data;
```

If you need to sort on several fields, the following paradigm is useful.

```
@sorted = sort { field1($a) <=> field1($b) ||
    field2($a) cmp field2($b) ||
    field3($a) cmp field3($b)
    } @data;
```

This can be conveniently combined with precalculation of keys as given above.

See the *sort* article in the "Far More Than You Ever Wanted To Know" collection in <http://www.cpan.org/olddoc/FMTEYEWTK.tgz> for more about this approach.

See also the question below on sorting hashes.

How do I manipulate arrays of bits?

Use `pack()` and `unpack()`, or else `vec()` and the bitwise operations.

For example, this sets `$vec` to have bit `N` set if `$ints[N]` was set:

```
$vec = '';
foreach(@ints) { vec($vec, $_, 1) = 1 }
```

Here's how, given a vector in `$vec`, you can get those bits into your `@ints` array:

```
sub bitvec_to_list {
    my $vec = shift;
    my @ints;
    # Find null-byte density then select best algorithm
    if ($vec =~ tr/\0// / length $vec > 0.95) {
        use integer;
        my $i;
        # This method is faster with mostly null-bytes
        while($vec =~ /[^\0]/g) {
            $i = -9 + 8 * pos $vec;
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
            push @ints, $i if vec($vec, ++$i, 1);
        }
    }
}
```

```

        push @ints, $i if vec($vec, ++$i, 1);
    }
} else {
    # This method is a fast general algorithm
    use integer;
    my $bits = unpack "b*", $vec;
    push @ints, 0 if $bits =~ s/^\(\\d)// && $1;
    push @ints, pos $bits while($bits =~ /1/g);
}
return \@ints;
}

```

This method gets faster the more sparse the bit vector is. (Courtesy of Tim Bunce and Winfried Koenig.)

You can make the while loop a lot shorter with this suggestion from Benjamin Goldberg:

```

while($vec =~ /[^\0]+/g ) {
    push @ints, grep vec($vec, $_, 1), $-[0] * 8 .. $+[0] * 8;
}

```

Or use the CPAN module Bit::Vector:

```

$vector = Bit::Vector->new($num_of_bits);
$vector->Index_List_Store(@ints);
@ints = $vector->Index_List_Read();

```

Bit::Vector provides efficient methods for bit vector, sets of small integers and "big int" math.

Here's a more extensive illustration using `vec()`:

```

# vec demo
$vector = "\xff\x0f\xef\xfe";
print "Ilya's string \\xff\\x0f\\xef\\xfe represents the number ",
    unpack("N", $vector), "\n";
$is_set = vec($vector, 23, 1);
print "Its 23rd bit is ", $is_set ? "set" : "clear", ".\n";
pvec($vector);

set_vec(1,1,1);
set_vec(3,1,1);
set_vec(23,1,1);

set_vec(3,1,3);
set_vec(3,2,3);
set_vec(3,4,3);
set_vec(3,4,7);
set_vec(3,8,3);
set_vec(3,8,7);

set_vec(0,32,17);
set_vec(1,32,17);

sub set_vec {
    my ($offset, $width, $value) = @_;
    my $vector = '';
    vec($vector, $offset, $width) = $value;
    print "offset=$offset width=$width value=$value\n";
    pvec($vector);
}

```

```

sub pvec {
    my $vector = shift;
    my $bits = unpack("b*", $vector);
    my $i = 0;
    my $BASE = 8;

    print "vector length in bytes: ", length($vector), "\n";
    @bytes = unpack("A8" x length($vector), $bits);
    print "bits are: @bytes\n\n";
}

```

Why does `defined()` return true on empty arrays and hashes?

The short story is that you should probably only use `defined` on scalars or functions, not on aggregates (arrays and hashes). See *defined* in the 5.004 release or later of Perl for more detail.

Data: Hashes (Associative Arrays)

How do I process an entire hash?

Use the `each()` function (see *each*) if you don't care whether it's sorted:

```

while ( ($key, $value) = each %hash) {
    print "$key = $value\n";
}

```

If you want it sorted, you'll have to use `foreach()` on the result of sorting the keys as shown in an earlier question.

What happens if I add or remove keys from a hash while iterating over it?

Don't do that. :-)

[lwall] In Perl 4, you were not allowed to modify a hash at all while iterating over it. In Perl 5 you can delete from it, but you still can't add to it, because that might cause a doubling of the hash table, in which half the entries get copied up to the new top half of the table, at which point you've totally bamboozled the iterator code. Even if the table doesn't double, there's no telling whether your new entry will be inserted before or after the current iterator position.

Either treasure up your changes and make them after the iterator finishes or use `keys` to fetch all the old keys at once, and iterate over the list of keys.

How do I look up a hash element by value?

Create a reverse hash:

```

%by_value = reverse %by_key;
$key = $by_value{$value};

```

That's not particularly efficient. It would be more space-efficient to use:

```

while (($key, $value) = each %by_key) {
    $by_value{$value} = $key;
}

```

If your hash could have repeated values, the methods above will only find one of the associated keys. This may or may not worry you. If it does worry you, you can always reverse the hash into a hash of arrays instead:

```

while (($key, $value) = each %by_key) {
    push @{$key_list_by_value{$value}}, $key;
}

```

How can I know how many entries are in a hash?

If you mean how many keys, then all you have to do is use the `keys()` function in a scalar context:

```
$num_keys = keys %hash;
```

The `keys()` function also resets the iterator, which means that you may see strange results if you use this between uses of other hash operators such as `each()`.

How do I sort a hash (optionally by value instead of key)?

Internally, hashes are stored in a way that prevents you from imposing an order on key–value pairs. Instead, you have to sort a list of the keys or values:

```
@keys = sort keys %hash;      # sorted by key
@keys = sort {
    $hash{$a} cmp $hash{$b}
} keys %hash;                # and by value
```

Here we'll do a reverse numeric sort by value, and if two keys are identical, sort by length of key, or if that fails, by straight ASCII comparison of the keys (well, possibly modified by your locale—see [perllocale](#)).

```
@keys = sort {
    $hash{$b} <=> $hash{$a}
    ||
    length($b) <=> length($a)
    ||
    $a cmp $b
} keys %hash;
```

How can I always keep my hash sorted?

You can look into using the `DB_File` module and `tie()` using the `$DB_BTREE` hash bindings as documented in [In Memory Databases in DB_File](#). The `Tie::IxHash` module from CPAN might also be instructive.

What's the difference between "delete" and "undef" with hashes?

Hashes contain pairs of scalars: the first is the key, the second is the value. The key will be coerced to a string, although the value can be any kind of scalar: string, number, or reference. If a key `$key` is present in `%hash`, `exists($hash{$key})` will return true. The value for a given key can be `undef`, in which case `$hash{$key}` will be `undef` while `exists $hash{$key}` will return true. This corresponds to `($key, undef)` being in the hash.

Pictures help... here's the `%hash` table:

keys	values
a	3
x	7
d	0
e	2

And these conditions hold

```
$hash{'a'}           is true
$hash{'d'}           is false
defined $hash{'d'}    is true
defined $hash{'a'}    is true
exists $hash{'a'}     is true (Perl5 only)
grep ($_ eq 'a', keys %hash) is true
```

If you now say

```
undef $hash{'a'}
```

your table now reads:

keys	values
a	undef
x	7
d	0
e	2

and these conditions now hold; changes in caps:

```
$hash{'a'}           is FALSE
$hash{'d'}           is false
defined $hash{'d'}    is true
defined $hash{'a'}    is FALSE
exists $hash{'a'}     is true (Perl5 only)
grep ($_ eq 'a', keys %hash) is true
```

Notice the last two: you have an undef value, but a defined key!

Now, consider this:

```
delete $hash{'a'}
```

your table now reads:

keys	values
x	7
d	0
e	2

and these conditions now hold; changes in caps:

```
$hash{'a'}           is false
$hash{'d'}           is false
defined $hash{'d'}    is true
defined $hash{'a'}    is false
exists $hash{'a'}     is FALSE (Perl5 only)
grep ($_ eq 'a', keys %hash) is FALSE
```

See, the whole entry is gone!

Why don't my tied hashes make the defined/exists distinction?

This depends on the tied hash's implementation of `EXISTS()`. For example, there isn't the concept of undef with hashes that are tied to DBM* files. It also means that `exists()` and `defined()` do the same thing with a DBM* file, and what they end up doing is not what they do with ordinary hashes.

How do I reset an `each()` operation part-way through?

Using `keys %hash` in scalar context returns the number of keys in the hash *and* resets the iterator associated with the hash. You may need to do this if you use `last` to exit a loop early so that when you re-enter it, the hash iterator has been reset.

How can I get the unique keys from two hashes?

First you extract the keys from the hashes into lists, then solve the "removing duplicates" problem described above. For example:

```
%seen = ();
for $element (keys(%foo), keys(%bar)) {
    $seen{$element}++;
}
@uniq = keys %seen;
```

Or more succinctly:

```
@uniq = keys %{ %foo,%bar };
```

Or if you really want to save space:

```
%seen = ();
while (defined ($key = each %foo)) {
    $seen{$key}++;
}
while (defined ($key = each %bar)) {
    $seen{$key}++;
}
@uniq = keys %seen;
```

How can I store a multidimensional array in a DBM file?

Either stringify the structure yourself (no fun), or else get the MLDBM (which uses Data::Dumper) module from CPAN and layer it on top of either DB_File or GDBM_File.

How can I make my hash remember the order I put elements into it?

Use the Tie::IxHash from CPAN.

```
use Tie::IxHash;
tie(%myhash, Tie::IxHash);
for ($i=0; $i<20; $i++) {
    $myhash{$i} = 2*$i;
}
@keys = keys %myhash;
# @keys = (0,1,2,3,...)
```

Why does passing a subroutine an undefined element in a hash create it?

If you say something like:

```
somefunc($hash{"nonesuch key here"});
```

Then that element "autovivifies"; that is, it springs into existence whether you store something there or not. That's because functions get scalars passed in by reference. If `somefunc()` modifies `$_[0]`, it has to be ready to write it back into the caller's version.

This has been fixed as of Perl5.004.

Normally, merely accessing a key's value for a nonexistent key does *not* cause that key to be forever there. This is different than `awk`'s behavior.

How can I make the Perl equivalent of a C structure/C++ class/hash or array of hashes or arrays?

Usually a hash ref, perhaps like this:

```
$record = {
    NAME    => "Jason",
    EMPNO   => 132,
    TITLE   => "deputy peon",
    AGE     => 23,
    SALARY  => 37_000,
    PALS    => [ "Norbert", "Rhys", "Phineas"],
};
```

References are documented in [perlref](#) and the upcoming [perlrefut](#). Examples of complex data structures are given in [perlisc](#) and [perllo](#). Examples of structures and object-oriented classes are in [perltoot](#).

How can I use a reference as a hash key?

You can't do this directly, but you could use the standard Tie::RefHash module distributed with Perl.

Data: Misc

How do I handle binary data correctly?

Perl is binary clean, so this shouldn't be a problem. For example, this works fine (assuming the files are found):

```
if (`cat /vmunix` =~ /gzip/) {
    print "Your kernel is GNU-zip enabled!\n";
}
```

On less elegant (read: Byzantine) systems, however, you have to play tedious games with "text" versus "binary" files. See [binmode in perlfunc](#) or [perlpentut](#). Most of these ancient-thinking systems are curses out of Microsoft, who seem to be committed to putting the backward into backward compatibility.

If you're concerned about 8-bit ASCII data, then see [perllocale](#).

If you want to deal with multibyte characters, however, there are some gotchas. See the section on Regular Expressions.

How do I determine whether a scalar is a number/whole/integer/float?

Assuming that you don't care about IEEE notations like "NaN" or "Infinity", you probably just want to use a regular expression.

```
if (/D/) { print "has nondigits\n" }
if (/^\d+$/) { print "is a whole number\n" }
if (/^-?\d+$/) { print "is an integer\n" }
if (/^[+-]?\d+$/) { print "is a +/- integer\n" }
if (/^-?\d+\.\d+$/) { print "is a real number\n" }
if (/^-?(?:\d+(?:\.\d*)?|\.\d+)\$/) { print "is a decimal number\n" }
if (/^[+-]?(?:\d+(\.\d*)?|\.\d+)\d*(\.\d*)?([Ee](?:[+-]?\d+)\d*)?\$/)
    { print "a C float\n" }
```

You can also use the [Data::Types/Data::Types](#) module on the CPAN, which exports functions that validate data types using these and other regular expressions.

If you're on a POSIX system, Perl's supports the `POSIX::strtod` function. Its semantics are somewhat cumbersome, so here's a `getnum` wrapper function for more convenient access. This function takes a string and returns the number it found, or `undef` for input that isn't a C float. The `is_numeric` function is a front end to `getnum` if you just want to say, "Is this a float?"

```
sub getnum {
    use POSIX qw(strtod);
    my $str = shift;
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;
    $! = 0;
    my($num, $unparsed) = strtod($str);
    if (($str eq '') || ($unparsed != 0) || $!) {
        return undef;
    } else {
        return $num;
    }
}
```



```
sub is_numeric { defined getnum($_[0]) }
```

Or you could check out the [String::Scanf](#)/[String::Scanf](#) module on the CPAN instead. The POSIX module (part of the standard Perl distribution) provides the `strtod` and `strtol` for converting strings to double and longs, respectively.

How do I keep persistent data across program calls?

For some specific applications, you can use one of the DBM modules. See [AnyDBM_File](#). More generically, you should consult the `FreezeThaw` or `Storable` modules from CPAN. Starting from Perl 5.8 `Storable` is part of the standard distribution. Here's one example using `Storable`'s `store` and `retrieve` functions:

```
use Storable;
store(\%hash, "filename");

# later on...
$href = retrieve("filename");      # by ref
%hash = %{ retrieve("filename") }; # direct to hash
```

How do I print out or copy a recursive data structure?

The `Data::Dumper` module on CPAN (or the 5.005 release of Perl) is great for printing out data structures. The `Storable` module, found on CPAN, provides a function called `dclone` that recursively copies its argument.

```
use Storable qw(dclone);
$r2 = dclone($r1);
```

Where `$r1` can be a reference to any kind of data structure you'd like. It will be deeply copied. Because `dclone` takes and returns references, you'd have to add extra punctuation if you had a hash of arrays that you wanted to copy.

```
%newhash = %{ dclone(\%oldhash) };
```

How do I define methods for every class/object?

Use the `UNIVERSAL` class (see [UNIVERSAL](#)).

How do I verify a credit card checksum?

Get the `Business::CreditCard` module from CPAN.

How do I pack arrays of doubles or floats for XS code?

The `kgbpack.c` code in the `PGPLOT` module on CPAN does just this. If you're doing a lot of float or double processing, consider using the `PDL` module from CPAN instead—it makes number-crunching easy.

AUTHOR AND COPYRIGHT

Copyright (c) 1997–2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

NAME

perlfaq5 – Files and Formats (\$Revision: 1.18 \$, \$Date: 2002/05/30 07:04:25 \$)

DESCRIPTION

This section deals with I/O and the "f" issues: filehandles, flushing, formats, and footers.

How do I flush/unbuffer an output filehandle? Why must I do this?

Perl does not support truly unbuffered output (except insofar as you can `syswrite(OUT, $char, 1)`), although it does support is "command buffering", in which a physical write is performed after every output command.

The C standard I/O library (`stdio`) normally buffers characters sent to devices so that there isn't a system call for each byte. In most `stdio` implementations, the type of output buffering and the size of the buffer varies according to the type of device. Perl's `print()` and `write()` functions normally buffer output, while `syswrite()` bypasses buffering all together.

If you want your output to be sent immediately when you execute `print()` or `write()` (for instance, for some network protocols), you must set the handle's autoflush flag. This flag is the Perl variable `$|` and when it is set to a true value, Perl will flush the handle's buffer after each `print()` or `write()`. Setting `$|` affects buffering only for the currently selected default file handle. You choose this handle with the one argument `select()` call (see [\\$|](#) and [select](#)).

Use `select()` to choose the desired handle, then set its per-filehandle variables.

```
$old_fh = select(OUTPUT_HANDLE);
$| = 1;
select($old_fh);
```

Some idioms can handle this in a single statement:

```
select((select(OUTPUT_HANDLE), $| = 1)[0]);
$| = 1, select $_ for select OUTPUT_HANDLE;
```

Some modules offer object-oriented access to handles and their variables, although they may be overkill if this is the only thing you do with them. You can use `IO::Handle`:

```
use IO::Handle;
open(DEV, ">/dev/printer"); # but is this?
DEV->autoflush(1);
```

or `IO::Socket`:

```
use IO::Socket; # this one is kinda a pipe?
my $sock = IO::Socket::INET->new( 'www.example.com:80' );

$sock->autoflush();
```

How do I change one line in a file/delete a line in a file/insert a line in the middle of a file/append to the beginning of a file?

Use the `Tie::File` module, which is included in the standard distribution since Perl 5.8.0.

How do I count the number of lines in a file?

One fairly efficient way is to count newlines in the file. The following program uses a feature of `tr///`, as documented in [perlop](#). If your text file doesn't end with a newline, then it's not really a proper text file, so this may report one fewer line than you expect.

```
$lines = 0;
open(FILE, $filename) or die "Can't open '$filename': $!";
while (sysread FILE, $buffer, 4096) {
    $lines += ($buffer =~ tr/\n//);
}
```

```
close FILE;
```

This assumes no funny games with newline translations.

How do I make a temporary file name?

Use the `File::Temp` module, see [File::Temp](#) for more information.

```
use File::Temp qw/ tempfile tempdir /;

$dir = tempdir( CLEANUP => 1 );
($fh, $filename) = tempfile( DIR => $dir );

# or if you don't need to know the filename

$fh = tempfile( DIR => $dir );
```

The `File::Temp` has been a standard module since Perl 5.6.1. If you don't have a modern enough Perl installed, use the `new_tmpfile` class method from the `IO::File` module to get a filehandle opened for reading and writing. Use it if you don't need to know the file's name:

```
use IO::File;
$fh = IO::File->new_tmpfile()
    or die "Unable to make new temporary file: $!";
```

If you're committed to creating a temporary file by hand, use the process ID and/or the current time-value. If you need to have many temporary files in one process, use a counter:

```
BEGIN {
    use Fcntl;
    my $temp_dir = -d '/tmp' ? '/tmp' : $ENV{TMPDIR} || $ENV{TEMP};
    my $base_name = sprintf("%s/%d-%d-0000", $temp_dir, $$, time());
    sub temp_file {
        local *FH;
        my $count = 0;
        until (defined(fileno(FH)) || $count++ > 100) {
            $base_name =~ s/-(\d+)$/"-". (1 + $1)/e;
            sysopen(FH, $base_name, O_WRONLY|O_EXCL|O_CREAT);
        }
        if (defined(fileno(FH)))
            return (*FH, $base_name);
        } else {
            return ();
        }
    }
}
```

How can I manipulate fixed-record-length files?

The most efficient way is using `pack()` and `unpack()`. This is faster than using `substr()` when taking many, many strings. It is slower for just a few.

Here is a sample chunk of code to break up and put back together again some fixed-format input lines, in this case from the output of a normal, Berkeley-style ps:

```
# sample input line:
# 15158 p5 T 0:00 perl /home/tchrist/scripts/now-what
$PS_T = 'A6 A4 A7 A5 A*';
open(PS, "ps|");
print scalar <PS>;
while (<PS>) {
    ($pid, $tt, $stat, $time, $command) = unpack($PS_T, $_);
    for $var (qw!pid tt stat time command!) {
```

```

        print "$var: <$$var>\n";
    }
    print 'line=', pack($PS_T, $pid, $tt, $stat, $time, $command),
        "\n";
}

```

We've used `$$var` in a way that is forbidden by `use strict 'refs'`. That is, we've promoted a string to a scalar variable reference using symbolic references. This is okay in small programs, but doesn't scale well.

It also only works on global variables, not lexicals.

How can I make a filehandle local to a subroutine? How do I pass filehandles between subroutines? How do I make an array of filehandles?

As of perl5.6, `open()` autovivifies file and directory handles as references if you pass it an uninitialized scalar variable. You can then pass these references just like any other scalar, and use them in the place of named handles.

```

open my $fh, $file_name;
open local $fh, $file_name;
print $fh "Hello World!\n";
process_file( $fh );

```

Before perl5.6, you had to deal with various typeglob idioms which you may see in older code.

```

open FILE, "> $filename";
process_typeglob( *FILE );
process_reference( \*FILE );

sub process_typeglob { local *FH = shift; print FH "Typeglob!" }
sub process_reference { local $fh = shift; print $fh "Reference!" }

```

If you want to create many anonymous handles, you should check out the `Symbol` or `IO::Handle` modules.

How can I use a filehandle indirectly?

An indirect filehandle is using something other than a symbol in a place that a filehandle is expected. Here are ways to get indirect filehandles:

```

$fh = SOME_FH;           # bareword is strict-subs hostile
$fh = "SOME_FH";         # strict-refs hostile; same package only
$fh = *SOME_FH;          # typeglob
$fh = \*SOME_FH;          # ref to typeglob (bless-able)
$fh = *SOME_FH{IO};      # blessed IO::Handle from *SOME_FH typeglob

```

Or, you can use the new method from one of the `IO::*` modules to create an anonymous filehandle, store that in a scalar variable, and use it as though it were a normal filehandle.

```

use IO::Handle;           # 5.004 or higher
$fh = IO::Handle->new();

```

Then use any of those as you would a normal filehandle. Anywhere that Perl is expecting a filehandle, an indirect filehandle may be used instead. An indirect filehandle is just a scalar variable that contains a filehandle. Functions like `print`, `open`, `seek`, or the `< <FH` diamond operator will accept either a named filehandle or a scalar variable containing one:

```

($ifh, $ofh, $efh) = (*STDIN, *STDOUT, *STDERR);
print $ofh "Type it: ";
$got = <$ifh>
print $efh "What was that: $got";

```

If you're passing a filehandle to a function, you can write the function in two ways:

```
sub accept_fh {
    my $fh = shift;
    print $fh "Sending to indirect filehandle\n";
}
```

Or it can localize a typeglob and use the filehandle directly:

```
sub accept_fh {
    local *FH = shift;
    print FH "Sending to localized filehandle\n";
}
```

Both styles work with either objects or typeglobs of real filehandles. (They might also work with strings under some circumstances, but this is risky.)

```
accept_fh(*STDOUT);
accept_fh($handle);
```

In the examples above, we assigned the filehandle to a scalar variable before using it. That is because only simple scalar variables, not expressions or subscripts of hashes or arrays, can be used with built-ins like `print`, `printf`, or the diamond operator. Using something other than a simple scalar variable as a filehandle is illegal and won't even compile:

```
@fd = (*STDIN, *STDOUT, *STDERR);
print $fd[1] "Type it: ";           # WRONG
$got = <$fd[0]>                     # WRONG
print $fd[2] "What was that: $got"; # WRONG
```

With `print` and `printf`, you get around this by using a block and an expression where you would place the filehandle:

```
print { $fd[1] } "funny stuff\n";
printf { $fd[1] } "Pity the poor %x.\n", 3_735_928_559;
# Pity the poor deadbeef.
```

That block is a proper block like any other, so you can put more complicated code there. This sends the message out to one of two places:

```
$ok = -x "/bin/cat";
print { $ok ? $fd[1] : $fd[2] } "cat stat $ok\n";
print { $fd[ 1+ ($ok || 0) ] } "cat stat $ok\n";
```

This approach of treating `print` and `printf` like object methods calls doesn't work for the diamond operator. That's because it's a real operator, not just a function with a comma-less argument. Assuming you've been storing typeglobs in your structure as we did above, you can use the built-in function named `readline` to read a record just as `<` does. Given the initialization shown above for `@fd`, this would work, but only because `readline()` requires a typeglob. It doesn't work with objects or strings, which might be a bug we haven't fixed yet.

```
$got = readline($fd[0]);
```

Let it be noted that the flakiness of indirect filehandles is not related to whether they're strings, typeglobs, objects, or anything else. It's the syntax of the fundamental operators. Playing the object game doesn't help you at all here.

How can I set up a footer format to be used with `write()`?

There's no builtin way to do this, but [perlform](#) has a couple of techniques to make it possible for the intrepid hacker.

How can I write() into a string?

See [Accessing Formatting Internals in perlform](#) for an `swrite()` function.

How can I output my numbers with commas added?

This one from Benjamin Goldberg will do it for you:

```
s/(^[+-]?\d+(?=(?>(?:\d{3})+)(?!\\d))|\\G\\d{3}(?==\\d))/ $1,/g;
```

or written verbosely:

```
s/(
    ^[+-]?           # beginning of number.
    \\d{1,3}?        # first digits before first comma
    (?=             # followed by, (but not included in the match) :
        (?>(?:\\d{3})+) # some positive multiple of three digits.
        (?!\\d)        # an *exact* multiple, not x * 3 + 1 or whatever.
    )
    |               # or:
    \\G\\d{3}       # after the last group, get three digits
    (?==\\d)        # but they have to have more digits after them.
)/ $1,/xg;
```

How can I translate tildes (~) in a filename?

Use the `<(glob())` operator, documented in [perlfunc](#). Older versions of Perl require that you have a shell installed that groks tildes. Recent perl versions have this feature built in. The `File::KGlob` module (available from CPAN) gives more portable glob functionality.

Within Perl, you may use this directly:

```
$filename =~ s{
    ^ ~             # find a leading tilde
    (               # save this in $1
        [^/]        # a non-slash character
        *           # repeated 0 or more times (0 means me)
    )
}{
    $1
    ? (getpwnam($1))[7]
    : ( $ENV{HOME} || $ENV{LOGDIR} )
}ex;
```

How come when I open a file read-write it wipes it out?

Because you're using something like this, which truncates the file and *then* gives you read-write access:

```
open(FH, "+> /path/name");           # WRONG (almost always)
```

Whoops. You should instead use this, which will fail if the file doesn't exist.

```
open(FH, "+< /path/name");           # open for update
```

Using `""` always clobbers or creates. Using `"<"` never does either. The `"+"` doesn't change this.

Here are examples of many kinds of file opens. Those using `sysopen()` all assume

```
use Fcntl;
```

To open file for reading:

```
open(FH, "< $path")                    || die $!;
sysopen(FH, $path, O_RDONLY)          || die $!;
```

To open file for writing, create new file if needed or else truncate old file:

```
open(FH, "> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT) || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT, 0666) || die $!;
```

To open file for writing, create new file, file must not exist:

```
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT) || die $!;
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT, 0666) || die $!;
```

To open file for appending, create if necessary:

```
open(FH, ">> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT) || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT, 0666) || die $!;
```

To open file for appending, file must exist:

```
sysopen(FH, $path, O_WRONLY|O_APPEND) || die $!;
```

To open file for update, file must exist:

```
open(FH, "+< $path") || die $!;
sysopen(FH, $path, O_RDWR) || die $!;
```

To open file for update, create file if necessary:

```
sysopen(FH, $path, O_RDWR|O_CREAT) || die $!;
sysopen(FH, $path, O_RDWR|O_CREAT, 0666) || die $!;
```

To open file for update, file must not exist:

```
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT) || die $!;
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT, 0666) || die $!;
```

To open a file without blocking, creating if necessary:

```
sysopen(FH, "/tmp/somefile", O_WRONLY|O_NDELAY|O_CREAT)
    or die "can't open /tmp/somefile: $!";
```

Be warned that neither creation nor deletion of files is guaranteed to be an atomic operation over NFS. That is, two processes might both successfully create or unlink the same file! Therefore `O_EXCL` isn't as exclusive as you might wish.

See also the new [perlopentut](#) if you have it (new for 5.6).

Why do I sometimes get an "Argument list too long" when I use <*>?

The `< <` operator performs a globbing operation (see above). In Perl versions earlier than v5.6.0, the internal `glob()` operator forks `csh(1)` to do the actual glob expansion, but `csh` can't handle more than 127 items and so gives the error message `Argument list too long`. People who installed `tcsh` as `csh` won't have this problem, but their users may be surprised by it.

To get around this, either upgrade to Perl v5.6.0 or later, do the glob yourself with `readdir()` and patterns, or use a module like `File::KGlob`, one that doesn't use the shell to do globbing.

Is there a leak/bug in `glob()`?

Due to the current implementation on some operating systems, when you use the `glob()` function or its angle-bracket alias in a scalar context, you may cause a memory leak and/or unpredictable behavior. It's best therefore to use `glob()` only in list context.

How can I open a file with a leading "" or trailing blanks?

Normally perl ignores trailing blanks in filenames, and interprets certain leading characters (or a trailing `"|"`) to mean something special.

The three argument form of `open()` lets you specify the mode separately from the filename. The `open()` function treats special mode characters and whitespace in the filename as literals

```
open FILE, "<", " file "; # filename is " file "
open FILE, ">", ">file"; # filename is ">file"
```

It may be a lot clearer to use `sysopen()`, though:

```
use Fcntl;
$badpath = "<<<something really wicked ";
sysopen (FH, $badpath, O_WRONLY | O_CREAT | O_TRUNC)
or die "can't open $badpath: $!";
```

How can I reliably rename a file?

If your operating system supports a proper `mv(1)` utility or its functional equivalent, this works:

```
rename($old, $new) or system("mv", $old, $new);
```

It may be more portable to use the `File::Copy` module instead. You just copy to the new file to the new name (checking return values), then delete the old one. This isn't really the same semantically as a `rename()`, which preserves meta-information like permissions, timestamps, inode info, etc.

Newer versions of `File::Copy` export a `move()` function.

How can I lock a file?

Perl's builtin `flock()` function (see [perlfunc](#) for details) will call `flock(2)` if that exists, `fcntl(2)` if it doesn't (on perl version 5.004 and later), and `lockf(3)` if neither of the two previous system calls exists. On some systems, it may even use a different form of native locking. Here are some gotchas with Perl's `flock()`:

- 1 Produces a fatal error if none of the three system calls (or their close equivalent) exists.
- 2 `lockf(3)` does not provide shared locking, and requires that the filehandle be open for writing (or appending, or read/writing).
- 3 Some versions of `flock()` can't lock files over a network (e.g. on NFS file systems), so you'd need to force the use of `fcntl(2)` when you build Perl. But even this is dubious at best. See the `flock` entry of [perlfunc](#) and the *INSTALL* file in the source distribution for information on building Perl to do this.

Two potentially non-obvious but traditional flock semantics are that it waits indefinitely until the lock is granted, and that its locks are *merely advisory*. Such discretionary locks are more flexible, but offer fewer guarantees. This means that files locked with `flock()` may be modified by programs that do not also use `flock()`. Cars that stop for red lights get on well with each other, but not with cars that don't stop for red lights. See the `perlport` manpage, your port's specific documentation, or your system-specific local manpages for details. It's best to assume traditional behavior if you're writing portable programs. (If you're not, you should as always feel perfectly free to write for your own system's idiosyncrasies (sometimes called "features"). Slavish adherence to portability concerns shouldn't get in the way of your getting your job done.)

For more information on file locking, see also [File Locking in perlport](#) if you have it (new for 5.6).

Why can't I just open(FH, "file.lock")?

A common bit of code **NOT TO USE** is this:

```
sleep(3) while -e "file.lock"; # PLEASE DO NOT USE
open(LCK, "> file.lock"); # THIS BROKEN CODE
```

This is a classic race condition: you take two steps to do something which must be done in one. That's why computer hardware provides an atomic test-and-set instruction. In theory, this "ought" to work:

```
sysopen(FH, "file.lock", O_WRONLY|O_EXCL|O_CREAT)
or die "can't open file.lock: $!";
```


except that lamentably, file creation (and deletion) is not atomic over NFS, so this won't work (at least, not every time) over the net. Various schemes involving `link()` have been suggested, but these tend to involve busy-wait, which is also subdesirable.

I still don't get locking. I just want to increment the number in the file. How can I do this?

Didn't anyone ever tell you web-page hit counters were useless? They don't count number of hits, they're a waste of time, and they serve only to stroke the writer's vanity. It's better to pick a random number; they're more realistic.

Anyway, this is what you can do if you can't help yourself.

```
use Fcntl qw(:DEFAULT :flock);
sysopen(FH, "numfile", O_RDWR|O_CREAT)      or die "can't open numfile: $!";
flock(FH, LOCK_EX)                          or die "can't flock numfile: $!";
$num = <FH> || 0;
seek(FH, 0, 0)                              or die "can't rewind numfile: $!";
truncate(FH, 0)                             or die "can't truncate numfile: $!";
(print FH $num+1, "\n")                     or die "can't write numfile: $!";
close FH                                    or die "can't close numfile: $!";
```

Here's a much better web-page hit counter:

```
$hits = int( (time() - 850_000_000) / rand(1_000) );
```

If the count doesn't impress your friends, then the code might. :-)

All I want to do is append a small amount of text to the end of a file. Do I still have to use locking?

If you are on a system that correctly implements `flock()` and you use the example appending code from "perldoc -f flock" everything will be OK even if the OS you are on doesn't implement append mode correctly (if such a system exists.) So if you are happy to restrict yourself to OSs that implement `flock()` (and that's not really much of a restriction) then that is what you should do.

If you know you are only going to use a system that does correctly implement appending (i.e. not Win32) then you can omit the `seek()` from the above code.

If you know you are only writing code to run on an OS and filesystem that does implement append mode correctly (a local filesystem on a modern Unix for example), and you keep the file in block-buffered mode and you write less than one buffer-full of output between each manual flushing of the buffer then each bufferload is almost guaranteed to be written to the end of the file in one chunk without getting intermingled with anyone else's output. You can also use the `syswrite()` function which is simply a wrapper around your systems `write(2)` system call.

There is still a small theoretical chance that a signal will interrupt the system level `write()` operation before completion. There is also a possibility that some STDIO implementations may call multiple system level `write()`s even if the buffer was empty to start. There may be some systems where this probability is reduced to zero.

How do I randomly update a binary file?

If you're just trying to patch a binary, in many cases something as simple as this works:

```
perl -i -pe 's{window manager}{window mangler}g' /usr/bin/emacs
```

However, if you have fixed sized records, then you might do something more like this:

```
$RECSIZE = 220; # size of record, in bytes
$recno   = 37;  # which record to update
open(FH, "+<somewhere") || die "can't update somewhere: $!";
seek(FH, $recno * $RECSIZE, 0);
read(FH, $record, $RECSIZE) == $RECSIZE || die "can't read record $recno: $!";
# munge the record
```

```
seek(FH, -$RECSIZE, 1);
print FH $record;
close FH;
```

Locking and error checking are left as an exercise for the reader. Don't forget them or you'll be quite sorry.

How do I get a file's timestamp in perl?

If you want to retrieve the time at which the file was last read, written, or had its meta-data (owner, etc) changed, you use the `-M`, `-A`, or `-C` file test operations as documented in [perlfunc](#). These retrieve the age of the file (measured against the start-time of your program) in days as a floating point number. Some platforms may not have all of these times. See [perlport](#) for details. To retrieve the "raw" time in seconds since the epoch, you would call the `stat` function, then use `localtime()`, `gmtime()`, or `POSIX::strftime()` to convert this into human-readable form.

Here's an example:

```
$write_secs = (stat($file))[9];
printf "file %s updated at %s\n", $file,
    scalar localtime($write_secs);
```

If you prefer something more legible, use the `File::stat` module (part of the standard distribution in version 5.004 and later):

```
# error checking left as an exercise for reader.
use File::stat;
use Time::localtime;
$date_string = ctime(stat($file)->mtime);
print "file $file updated at $date_string\n";
```

The `POSIX::strftime()` approach has the benefit of being, in theory, independent of the current locale. See [perllocale](#) for details.

How do I set a file's timestamp in perl?

You use the `utime()` function documented in [utime](#). By way of example, here's a little program that copies the read and write times from its first argument to all the rest of them.

```
if (@ARGV < 2) {
    die "usage: cptimes timestamp_file other_files ...\n";
}
$timestamp = shift;
($atime, $mtime) = (stat($timestamp))[8,9];
utime $atime, $mtime, @ARGV;
```

Error checking is, as usual, left as an exercise for the reader.

Note that `utime()` currently doesn't work correctly with Win95/NT ports. A bug has been reported. Check it carefully before using `utime()` on those platforms.

How do I print to more than one file at once?

If you only have to do this once, you can do this:

```
for $fh (FH1, FH2, FH3) { print $fh "whatever\n" }
```

To connect up to one filehandle to several output filehandles, it's easiest to use the `tee(1)` program if you have it, and let it take care of the multiplexing:

```
open (FH, "| tee file1 file2 file3");
```

Or even:

```
# make STDOUT go to three files, plus original STDOUT
open (STDOUT, "| tee file1 file2 file3") or die "Teeing off: $!\n";
```

```
print "whatever\n"           or die "Writing: $!\n";
close(STDOUT)                or die "Closing: $!\n";
```

Otherwise you'll have to write your own multiplexing print function—or your own tee program—or use Tom Christiansen's, at <http://www.cpan.org/authors/id/TOMC/scripts/tct.gz>, which is written in Perl and offers much greater functionality than the stock version.

How can I read in an entire file all at once?

The customary Perl approach for processing all the lines in a file is to do so one line at a time:

```
open (INPUT, $file)          || die "can't open $file: $!";
while (<INPUT>) {
    chomp;
    # do something with $_
}
close(INPUT)                  || die "can't close $file: $!";
```

This is tremendously more efficient than reading the entire file into memory as an array of lines and then processing it one element at a time, which is often—if not almost always—the wrong approach. Whenever you see someone do this:

```
@lines = <INPUT>;
```

you should think long and hard about why you need everything loaded at once. It's just not a scalable solution. You might also find it more fun to use the standard `Tie::File` module, or the `DB_File` module's `$DB_RECNO` bindings, which allow you to tie an array to a file so that accessing an element the array actually accesses the corresponding line in the file.

You can read the entire filehandle contents into a scalar.

```
{
    local(*INPUT, $/);
    open (INPUT, $file)      || die "can't open $file: $!";
    $var = <INPUT>;
}
```

That temporarily undefs your record separator, and will automatically close the file at block exit. If the file is already open, just use this:

```
$var = do { local $/; <INPUT> };
```

For ordinary files you can also use the `read` function.

```
read( INPUT, $var, -s INPUT );
```

The third argument tests the byte size of the data on the `INPUT` filehandle and reads that many bytes into the buffer `$var`.

How can I read in a file by paragraphs?

Use the `$/` variable (see [perlvar](#) for details). You can either set it to `"` to eliminate empty paragraphs (`"abc\n\n\n\ndef"`, for instance, gets treated as two paragraphs and not three), or `"\n\n"` to accept empty paragraphs.

Note that a blank line must have no blanks in it. Thus

`"fred\n \nstuff\n\n"` is one paragraph, but `"fred\n\nstuff\n\n"` is two.

How can I read a single character from a file? From the keyboard?

You can use the builtin `getc()` function for most filehandles, but it won't (easily) work on a terminal device. For `STDIN`, either use the `Term::ReadKey` module from CPAN or use the sample code in [getc](#).

If your system supports the portable operating system programming interface (POSIX), you can use the following code, which you'll note turns off echo processing as well.

```
#!/usr/bin/perl -w
use strict;
$| = 1;
for (1..4) {
    my $got;
    print "gimme: ";
    $got = getone();
    print "--> $got\n";
}
exit;

BEGIN {
    use POSIX qw(:termios_h);

    my ($term, $oterm, $echo, $noecho, $fd_stdin);

    $fd_stdin = fileno(STDIN);

    $term      = POSIX::Termios->new();
    $term->getattr($fd_stdin);
    $oterm     = $term->getlflag();

    $echo      = ECHO | ECHOK | ICANON;
    $noecho    = $oterm & ~$echo;

    sub cbreak {
        $term->setlflag($noecho);
        $term->setcc(VTIME, 1);
        $term->setattr($fd_stdin, TCSANOW);
    }

    sub cooked {
        $term->setlflag($oterm);
        $term->setcc(VTIME, 0);
        $term->setattr($fd_stdin, TCSANOW);
    }

    sub getone {
        my $key = '';
        cbreak();
        sysread(STDIN, $key, 1);
        cooked();
        return $key;
    }
}

END { cooked() }
```

The `Term::ReadKey` module from CPAN may be easier to use. Recent versions include also support for non-portable systems as well.

```
use Term::ReadKey;
open(TTY, "</dev/tty");
print "Gimme a char: ";
ReadMode "raw";
$key = ReadKey 0, *TTY;
ReadMode "normal";
printf "\nYou said %s, char number %03d\n",
    $key, ord $key;
```

How can I tell whether there's a character waiting on a filehandle?

The very first thing you should do is look into getting the Term::ReadKey extension from CPAN. As we mentioned earlier, it now even has limited support for non-portable (read: not open systems, closed, proprietary, not POSIX, not Unix, etc) systems.

You should also check out the Frequently Asked Questions list in comp.unix.* for things like this: the answer is essentially the same. It's very system dependent. Here's one solution that works on BSD systems:

```
sub key_ready {
    my($rin, $nfd);
    vec($rin, fileno(STDIN), 1) = 1;
    return $nfd = select($rin, undef, undef, 0);
}
```

If you want to find out how many characters are waiting, there's also the FIONREAD ioctl call to be looked at. The *h2ph* tool that comes with Perl tries to convert C include files to Perl code, which can be required. FIONREAD ends up defined as a function in the *sys/ioctl.ph* file:

```
require 'sys/ioctl.ph';

$size = pack("L", 0);
ioctl(FH, FIONREAD(), $size)    or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

If *h2ph* wasn't installed or doesn't work for you, you can *grep* the include files by hand:

```
% grep FIONREAD /usr/include/*/*
/usr/include/asm/ioctls.h:#define FIONREAD    0x541B
```

Or write a small C program using the editor of champions:

```
% cat > fionread.c
#include <sys/ioctl.h>
main() {
    printf("%#08x\n", FIONREAD);
}
^D
% cc -o fionread fionread.c
% ./fionread
0x4004667f
```

And then hard code it, leaving porting as an exercise to your successor.

```
$FIONREAD = 0x4004667f;          # XXX: opsys dependent

$size = pack("L", 0);
ioctl(FH, $FIONREAD, $size)    or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

FIONREAD requires a filehandle connected to a stream, meaning that sockets, pipes, and tty devices work, but *not* files.

How do I do a `tail -f` in perl?

First try

```
seek(GWFILE, 0, 1);
```

The statement `seek(GWFILE, 0, 1)` doesn't change the current position, but it does clear the end-of-file condition on the handle, so that the next `<GWFILE` makes Perl try again to read something.

If that doesn't work (it relies on features of your stdio implementation), then you need something more like this:

```

    for (;;) {
        for ($curpos = tell(GWFILE); <GWFILE>; $curpos = tell(GWFILE)) {
            # search for some stuff and put it into files
        }
        # sleep for a while
        seek(GWFILE, $curpos, 0); # seek to where we had been
    }

```

If this still doesn't work, look into the POSIX module. POSIX defines the `clearerr()` method, which can remove the end of file condition on a filehandle. The method: `read` until end of file, `clearerr()`, `read` some more. Lather, rinse, repeat.

There's also a `File::Tail` module from CPAN.

How do I dup() a filehandle in Perl?

If you check [open](#), you'll see that several of the ways to call `open()` should do the trick. For example:

```

open(LOG, ">>/tmp/logfile");
open(STDERR, ">&LOG");

```

Or even with a literal numeric descriptor:

```

$fd = $ENV{MHCONTEXTFD};
open(MHCONTEXT, "<&=$fd"); # like fdopen(3S)

```

Note that "<&STDIN" makes a copy, but "<=&STDIN" make an alias. That means if you close an aliased handle, all aliases become inaccessible. This is not true with a copied one.

Error checking, as always, has been left as an exercise for the reader.

How do I close a file descriptor by number?

This should rarely be necessary, as the Perl `close()` function is to be used for things that Perl opened itself, even if it was a dup of a numeric descriptor as with MHCONTEXT above. But if you really have to, you may be able to do this:

```

require 'sys/syscall.ph';
$rc = syscall(&SYS_close, $fd + 0); # must force numeric
die "can't sysclose $fd: $!" unless $rc == -1;

```

Or, just use the `fdopen(3S)` feature of `open()`:

```

{
    local *F;
    open F, "<&=$fd" or die "Cannot reopen fd=$fd: $!";
    close F;
}

```

Why can't I use "C:\temp\foo" in DOS paths? Why doesn't 'C:\temp\foo.exe' work?

Whoops! You just put a tab and a formfeed into that filename! Remember that within double quoted strings ("like this"), the backslash is an escape character. The full list of these is in [Quote and Quote-like Operators](#). Unsurprisingly, you don't have a file called "c:(tab)emp(formfeed)oo" or "c:(tab)emp(formfeed)oo.exe" on your legacy DOS filesystem.

Either single-quote your strings, or (preferably) use forward slashes. Since all DOS and Windows versions since something like MS-DOS 2.0 or so have treated / and \ the same in a path, you might as well use the one that doesn't clash with Perl—or the POSIX shell, ANSI C and C++, awk, Tcl, Java, or Python, just to mention a few. POSIX paths are more portable, too.

Why doesn't glob("*.") get all the files?

Because even on non-Unix ports, Perl's `glob` function follows standard Unix globbing semantics. You'll need `glob("*")` to get all (non-hidden) files. This makes `glob()` portable even to legacy systems.

Your port may include proprietary globbing functions as well. Check its documentation for details.

Why does Perl let me delete read-only files? Why does `-i` clobber protected files? Isn't this a bug in Perl?

This is elaborately and painstakingly described in the *file-dir-perms* article in the "Far More Than You Ever Wanted To Know" collection in <http://www.cpan.org/olddoc/FMTEYEWTK.tgz>.

The executive summary: learn how your filesystem works. The permissions on a file say what can happen to the data in that file. The permissions on a directory say what can happen to the list of files in that directory. If you delete a file, you're removing its name from the directory (so the operation depends on the permissions of the directory, not of the file). If you try to write to the file, the permissions of the file govern whether you're allowed to.

How do I select a random line from a file?

Here's an algorithm from the Camel Book:

```
srand;
rand($.) < 1 && ($line = $_) while <>;
```

This has a significant advantage in space over reading the whole file in. A simple proof by induction is available upon request if you doubt the algorithm's correctness.

Why do I get weird spaces when I print an array of lines?

Saying

```
print "@lines\n";
```

joins together the elements of `@lines` with a space between them. If `@lines` were `("little", "fluffy", "clouds")` then the above statement would print

```
little fluffy clouds
```

but if each element of `@lines` was a line of text, ending a newline character `("little\n", "fluffy\n", "clouds\n")` then it would print:

```
little
fluffy
clouds
```

If your array contains lines, just print them:

```
print @lines;
```

AUTHOR AND COPYRIGHT

Copyright (c) 1997–2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples here are in the public domain. You are permitted and encouraged to use this code and any derivatives thereof in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit to the FAQ would be courteous but is not required.

NAME

perlfaq6 – Regular Expressions (\$Revision: 1.12 \$, \$Date: 2002/06/01 22:31:09 \$)

DESCRIPTION

This section is surprisingly small because the rest of the FAQ is littered with answers involving regular expressions. For example, decoding a URL and checking whether something is a number are handled with regular expressions, but those answers are found elsewhere in this document (in [perlfaq9](#): “How do I decode or create those %-encodings on the web” and [perlfaq4](#): “How do I determine whether a scalar is a number/whole/integer/float”, to be precise).

How can I hope to use regular expressions without creating illegible and unmaintainable code?

Three techniques can make regular expressions maintainable and understandable.

Comments Outside the Regex

Describe what you’re doing and how you’re doing it, using normal Perl comments.

```
# turn the line into the first word, a colon, and the
# number of characters on the rest of the line
s/^(\\w+) (.*)/ lc($1) . ":" . length($2) /meg;
```

Comments Inside the Regex

The `/x` modifier causes whitespace to be ignored in a regex pattern (except in a character class), and also allows you to use normal comments there, too. As you can imagine, whitespace and comments help a lot.

`/x` lets you turn this:

```
s{<(?: [^>'"] * | ".*" | '.*?') +>}{}gs;
```

into this:

```
s{ <                                # opening angle bracket
  (?:                                # Non-backreffing grouping paren
    [^>'"] *                        # 0 or more things that are neither > nor ' nor "
    |                                # or else
    ".*"                            # a section between double quotes (stingy match)
    |                                # or else
    '.*?'                          # a section between single quotes (stingy match)
  ) +                                # all occurring one or more times
  >                                # closing angle bracket
}{}gsx;                            # replace with nothing, i.e. delete
```

It’s still not quite so clear as prose, but it is very useful for describing the meaning of each part of the pattern.

Different Delimiters

While we normally think of patterns as being delimited with `/` characters, they can be delimited by almost any character. [perlre](#) describes this. For example, the `s///` above uses braces as delimiters. Selecting another delimiter can avoid quoting the delimiter within the pattern:

```
s\\/usr\\/local\\/usr\\/share/g;      # bad delimiter choice
s#/usr/local#/usr/share#g;          # better
```

I’m having trouble matching over more than one line. What’s wrong?

Either you don’t have more than one line in the string you’re looking at (probably), or else you aren’t using the correct modifier(s) on your pattern (possibly).

There are many ways to get multiline data into a string. If you want it to happen automatically while reading input, you’ll want to set `$/` (probably to `''` for paragraphs or `undef` for the whole file) to allow you to read more than one line at a time.

Read [perlre](#) to help you decide which of `/s` and `/m` (or both) you might want to use: `/s` allows dot to include newline, and `/m` allows caret and dollar to match next to a newline, not just at the end of the string. You do need to make sure that you've actually got a multiline string in there.

For example, this program detects duplicate words, even when they span line breaks (but not paragraph ones). For this example, we don't need `/s` because we aren't using dot in a regular expression that we want to cross line boundaries. Neither do we need `/m` because we aren't wanting caret or dollar to match at any point inside the record next to newlines. But it's imperative that `$/` be set to something other than the default, or else we won't actually ever have a multiline record read in.

```
$/ = '';                # read in more whole paragraph, not just one line
while ( <> ) {
    while ( /\b([\w'-]+)(\s+\l)+\b/gi ) { # word starts alpha
        print "Duplicate $1 at paragraph $.\n";
    }
}
```

Here's code that finds sentences that begin with "From " (which would be mangled by many mailers):

```
$/ = '';                # read in more whole paragraph, not just one line
while ( <> ) {
    while ( /^From /gm ) { # /m makes ^ match next to \n
        print "leading from in paragraph $.\n";
    }
}
```

Here's code that finds everything between START and END in a paragraph:

```
undef $/;                # read in whole file, not just one line or paragraph
while ( <> ) {
    while ( /START(.*)END/sgm ) { # /s makes . cross line boundaries
        print "$1\n";
    }
}
```

How can I pull out lines between two patterns that are themselves on different lines?

You can use Perl's somewhat exotic `..` operator (documented in [perlop](#)):

```
perl -ne 'print if /START/ .. /END/' file1 file2 ...
```

If you wanted text and not lines, you would use

```
perl -0777 -ne 'print "$1\n" while /START(.*)END/gs' file1 file2 ...
```

But if you want nested occurrences of START through END, you'll run up against the problem described in the question in this section on matching balanced text.

Here's another example of using `..`:

```
while ( <> ) {
    $in_header = 1 .. /^$/;
    $in_body   = /^$/ .. eof();
    # now choose between them
} continue {
    reset if eof();          # fix $.
}
```

I put a regular expression into `$/` but it didn't work. What's wrong?

`$/` must be a string, not a regular expression. Awk has to be better for something. :-)

Actually, you could do this if you don't mind reading the whole file into memory:

```
undef $/;
@records = split /your_pattern/, <FH>;
```

The Net::Telnet module (available from CPAN) has the capability to wait for a pattern in the input stream, or timeout if it doesn't appear within a certain time.

```
## Create a file with three lines.
open FH, ">file";
print FH "The first line\nThe second line\nThe third line\n";
close FH;

## Get a read/write filehandle to it.
$fh = new IO::File "+<file";

## Attach it to a "stream" object.
use Net::Telnet;
$file = new Net::Telnet (-fhopen => $fh);

## Search for the second line and print out the third.
$file->waitfor('/second line\n/');
print $file->getline;
```

How do I substitute case insensitively on the LHS while preserving case on the RHS?

Here's a lovely Perlish solution by Larry Rosler. It exploits properties of bitwise xor on ASCII strings.

```
$_ = "this is a TEST case";

$old = 'test';
$new = 'success';

s{(\Q$old\E)}
{ uc $new | (uc $1 ^ $1) .
  (uc(substr $1, -1) ^ substr $1, -1) x
  (length($new) - length $1)
}egi;

print;
```

And here it is as a subroutine, modeled after the above:

```
sub preserve_case($$) {
    my ($old, $new) = @_;
    my $mask = uc $old ^ $old;

    uc $new | $mask .
        substr($mask, -1) x (length($new) - length($old))
}

$a = "this is a TEST case";
$a =~ s/(test)/preserve_case($1, "success")/egi;
print "$a\n";
```

This prints:

```
this is a SUcCESS case
```

As an alternative, to keep the case of the replacement word if it is longer than the original, you can use this code, by Jeff Pinyan:

```
sub preserve_case {
    my ($from, $to) = @_;
    my ($lf, $lt) = map length, @_;
```

```

    if ($lt < $lf) { $from = substr $from, 0, $lt }
    else { $from .= substr $to, $lf }

    return uc $to | ($from ^ uc $from);
}

```

This changes the sentence to "this is a SUCcess case."

Just to show that C programmers can write C in any programming language, if you prefer a more C-like solution, the following script makes the substitution have the same case, letter by letter, as the original. (It also happens to run about 240% slower than the Perlsh solution runs.) If the substitution has more characters than the string being substituted, the case of the last character is used for the rest of the substitution.

```

# Original by Nathan Torkington, massaged by Jeffrey Friedl
#
sub preserve_case($$)
{
    my ($old, $new) = @_;
    my ($state) = 0; # 0 = no change; 1 = lc; 2 = uc
    my ($i, $oldlen, $newlen, $c) = (0, length($old), length($new));
    my ($len) = $oldlen < $newlen ? $oldlen : $newlen;

    for ($i = 0; $i < $len; $i++) {
        if ($c = substr($old, $i, 1), $c =~ /[\\W\\d_]/) {
            $state = 0;
        } elsif (lc $c eq $c) {
            substr($new, $i, 1) = lc(substr($new, $i, 1));
            $state = 1;
        } else {
            substr($new, $i, 1) = uc(substr($new, $i, 1));
            $state = 2;
        }
    }
    # finish up with any remaining new (for when new is longer than old)
    if ($newlen > $oldlen) {
        if ($state == 1) {
            substr($new, $oldlen) = lc(substr($new, $oldlen));
        } elsif ($state == 2) {
            substr($new, $oldlen) = uc(substr($new, $oldlen));
        }
    }
    return $new;
}

```

How can I make `\w` match national character sets?

See [perllocale](#).

How can I match a locale-smart version of `/[a-zA-Z]/?`

One alphabetic character would be `/[^\W\d_]/`, no matter what locale you're in. Non-alphabetic would be `/[\\W\\d_]/` (assuming you don't consider an underscore a letter).

How can I quote a variable to use in a regex?

The Perl parser will expand `$variable` and `@variable` references in regular expressions unless the delimiter is a single quote. Remember, too, that the right-hand side of a `s///` substitution is considered a double-quoted string (see [perlop](#) for more details). Remember also that any regex special characters will be acted on unless you precede the substitution with `\Q`. Here's an example:

```
$string = "to die?";
```

```
$lhs = "die?";
$rhs = "sleep, no more";

$string =~ s/\Q$lhs/$rhs/;
# $string is now "to sleep no more"
```

Without the `\Q`, the regex would also spuriously match "di".

What is `/o` really for?

Using a variable in a regular expression match forces a re-evaluation (and perhaps recompilation) each time the regular expression is encountered. The `/o` modifier locks in the regex the first time it's used. This always happens in a constant regular expression, and in fact, the pattern was compiled into the internal format at the same time your entire program was.

Use of `/o` is irrelevant unless variable interpolation is used in the pattern, and if so, the regex engine will neither know nor care whether the variables change after the pattern is evaluated the *very first* time.

`/o` is often used to gain an extra measure of efficiency by not performing subsequent evaluations when you know it won't matter (because you know the variables won't change), or more rarely, when you don't want the regex to notice if they do.

For example, here's a "paragrep" program:

```
$/ = ''; # paragraph mode
$pat = shift;
while (<>) {
    print if $pat/o;
}
```

How do I use a regular expression to strip C style comments from a file?

While this actually can be done, it's much harder than you'd think. For example, this one-liner

```
perl -0777 -pe 's{/\*.*?\*/}{ }gs' foo.c
```

will work in many but not all cases. You see, it's too simple-minded for certain kinds of C programs, in particular, those with what appear to be comments in quoted strings. For that, you'd need something like this, created by Jeffrey Friedl and later modified by Fred Curtis.

```
$/ = undef;
$_ = <>;
s#/\* [^*] * \+ ( [^/*] [^*] * \+ ) * / ( " ( \\. | [^"\\] ) * " | ' ( \\. | [^'\\] ) * ' | . [^/"'\\] * ) # $2 #gs
print;
```

This could, of course, be more legibly written with the `/x` modifier, adding whitespace and comments. Here it is expanded, courtesy of Fred Curtis.

```
s{
    /\*          ## Start of /* ... */ comment
    [^*] * \+    ## Non-* followed by 1-or-more *'s
    (
        [^/*] [^*] * \+
    ) *          ## 0-or-more things which don't start with /
                ## but do end with '*'
    /           ## End of /* ... */ comment
|              ## OR various things which aren't comments:
    (
        "        ## Start of " ... " string
        (
            \\.    ## Escaped char
            |      ## OR
```

```

        [^"\\]      ## Non "\
    ) *
    "              ## End of " ... " string
|
    ## OR
    '              ## Start of ' ... ' string
(
    \\            ## Escaped char
|
    ## OR
    [^'\\]        ## Non '\
) *
'                ## End of ' ... ' string
|
    ## OR
    .              ## Anything other char
    [^/"'\\] *    ## Chars which doesn't start a comment, string or escape
)
}{ $2 } gxs;

```

A slight modification also removes C++ comments:

```
s#/\* [^*] * \+ ( [^/*] [^*] * \+ ) * /|// [^\n] * | (" (\\\. | [^"\\]) * " | ' (\\\. | [^'\\]) * ' | \. [^/"'\\]
```

Can I use Perl regular expressions to match balanced text?

Historically, Perl regular expressions were not capable of matching balanced text. As of more recent versions of perl including 5.6.1 experimental features have been added that make it possible to do this. Look at the documentation for the (??{ }) construct in recent perlre manual pages to see an example of matching balanced parentheses. Be sure to take special notice of the warnings present in the manual before making use of this feature.

CPAN contains many modules that can be useful for matching text depending on the context. Damian Conway provides some useful patterns in Regexp::Common. The module Text::Balanced provides a general solution to this problem.

One of the common applications of balanced text matching is working with XML and HTML. There are many modules available that support these needs. Two examples are HTML::Parser and XML::Parser. There are many others.

An elaborate subroutine (for 7-bit ASCII only) to pull out balanced and possibly nested single chars, like ` and ', { and }, or (and) can be found in http://www.cpan.org/authors/id/TOMC/scripts/pull_quotes.gz.

The C::Scan module from CPAN also contains such subs for internal use, but they are undocumented.

What does it mean that regexes are greedy? How can I get around it?

Most people mean that greedy regexes match as much as they can. Technically speaking, it's actually the quantifiers (?, *, +, { }) that are greedy rather than the whole pattern; Perl prefers local greed and immediate gratification to overall greed. To get non-greedy versions of the same quantifiers, use (??, *?, +?, { }?).

An example:

```

$s1 = $s2 = "I am very very cold";
$s1 =~ s/ve.*y //;      # I am cold
$s2 =~ s/ve.*?y //;     # I am very cold

```

Notice how the second substitution stopped matching as soon as it encountered "y ". The *? quantifier effectively tells the regular expression engine to find a match as quickly as possible and pass control on to whatever is next in line, like you would if you were playing hot potato.

How do I process each word on each line?

Use the split function:

```
while (<>) {
    foreach $word ( split ) {
        # do something with $word here
    }
}
```

Note that this isn't really a word in the English sense; it's just chunks of consecutive non-whitespace characters.

To work with only alphanumeric sequences (including underscores), you might consider

```
while (<>) {
    foreach $word (m/(\w+)/g) {
        # do something with $word here
    }
}
```

How can I print out a word-frequency or line-frequency summary?

To do this, you have to parse out each word in the input stream. We'll pretend that by word you mean chunk of alphabetic, hyphens, or apostrophes, rather than the non-whitespace chunk idea of a word given in the previous question:

```
while (<>) {
    while ( /(\b[^\W_\d][\w'-]+\b)/g ) { # misses "`sheep'"
        $seen{$1}++;
    }
}
while ( ($word, $count) = each %seen ) {
    print "$count $word\n";
}
```

If you wanted to do the same thing for lines, you wouldn't need a regular expression:

```
while (<>) {
    $seen{$_}++;
}
while ( ($line, $count) = each %seen ) {
    print "$count $line";
}
```

If you want these output in a sorted order, see [perlfaq4](#): “How do I sort a hash (optionally by value instead of key)?”.

How can I do approximate matching?

See the module `String::Approx` available from CPAN.

How do I efficiently match many regular expressions at once?

The following is extremely inefficient:

```
# slow but obvious way
@popstates = qw(CO ON MI WI MN);
while (defined($line = <>)) {
    for $state (@popstates) {
        if ($line =~ /\b$state\b/i) {
            print $line;
            last;
        }
    }
}
```

```

    }
  }
}

```

That's because Perl has to recompile all those patterns for each of the lines of the file. As of the 5.005 release, there's a much better approach, one which makes use of the new `qr//` operator:

```

# use spiffy new qr// operator, with /i flag even
use 5.005;
@popstates = qw(CO ON MI WI MN);
@poppats   = map { qr/\b$_\b/i } @popstates;
while (defined($line = <>)) {
    for $patobj (@poppats) {
        print $line if $line =~ /$patobj/;
    }
}

```

Why don't word-boundary searches with `\b` work for me?

Two common misconceptions are that `\b` is a synonym for `\s+` and that it's the edge between whitespace characters and non-whitespace characters. Neither is correct. `\b` is the place between a `\w` character and a `\W` character (that is, `\b` is the edge of a "word"). It's a zero-width assertion, just like `^`, `$`, and all the other anchors, so it doesn't consume any characters. [perlre](#) describes the behavior of all the regex metacharacters.

Here are examples of the incorrect application of `\b`, with fixes:

```

"two words" =~ /(\w+)\b(\w+)/;          # WRONG
"two words" =~ /(\w+)\s+(\w+)/;          # right

" =matchless= text" =~ /\b=(\w+)=\b/;    # WRONG
" =matchless= text" =~ /= (\w+) =/;      # right

```

Although they may not do what you thought they did, `\b` and `\B` can still be quite useful. For an example of the correct use of `\b`, see the example of matching duplicate words over multiple lines.

An example of using `\B` is the pattern `\Bis\b`. This will find occurrences of "is" on the insides of words only, as in "thistle", but not "this" or "island".

Why does using `$&`, `$'`, or `$'` slow my program down?

Once Perl sees that you need one of these variables anywhere in the program, it provides them on each and every pattern match. The same mechanism that handles these provides for the use of `$1`, `$2`, etc., so you pay the same price for each regex that contains capturing parentheses. If you never use `$&`, etc., in your script, then regexes *without* capturing parentheses won't be penalized. So avoid `$&`, `$'`, and `$'` if you can, but if you can't, once you've used them at all, use them at will because you've already paid the price. Remember that some algorithms really appreciate them. As of the 5.005 release, the `$&` variable is no longer "expensive" the way the other two are.

What good is `\G` in a regular expression?

The notation `\G` is used in a match or substitution in conjunction with the `/g` modifier to anchor the regular expression to the point just past where the last match occurred, i.e. the `pos()` point. A failed match resets the position of `\G` unless the `/c` modifier is in effect. `\G` can be used in a match without the `/g` modifier; it acts the same (i.e. still anchors at the `pos()` point) but of course only matches once and does not update `pos()`, as non-`/g` expressions never do. `\G` in an expression applied to a target string that has never been matched against a `/g` expression before or has had its `pos()` reset is functionally equivalent to `\A`, which matches at the beginning of the string.

For example, suppose you had a line of text quoted in standard mail and Usenet notation, (that is, with leading `<` characters), and you want change each leading `<` into a corresponding `:`. You could do so in this way:

```
s/^(>+)/': ' x length($1)/gem;
```

Or, using `\G`, the much simpler (and faster):

```
s/\G>/:/g;
```

A more sophisticated use might involve a tokenizer. The following lex-like example is courtesy of Jeffrey Friedl. It did not work in 5.003 due to bugs in that release, but does work in 5.004 or better. (Note the use of `/c`, which prevents a failed match with `/g` from resetting the search position back to the beginning of the string.)

```
while (<>) {
    chomp;
    PARSER: {
        m/ \G( \d+\b    )/gcx    && do { print "number: $1\n"; redo; };
        m/ \G( \w+      )/gcx    && do { print "word:   $1\n"; redo; };
        m/ \G( \s+      )/gcx    && do { print "space:  $1\n"; redo; };
        m/ \G( [^\w\d]+ )/gcx    && do { print "other:  $1\n"; redo; };
    }
}
```

Of course, that could have been written as

```
while (<>) {
    chomp;
    PARSER: {
        if ( /\G( \d+\b    )/gcx {
            print "number: $1\n";
            redo PARSER;
        }
        if ( /\G( \w+      )/gcx {
            print "word:   $1\n";
            redo PARSER;
        }
        if ( /\G( \s+      )/gcx {
            print "space:  $1\n";
            redo PARSER;
        }
        if ( /\G( [^\w\d]+ )/gcx {
            print "other:  $1\n";
            redo PARSER;
        }
    }
}
```

but then you lose the vertical alignment of the regular expressions.

Are Perl regexes DFAs or NFAs? Are they POSIX compliant?

While it's true that Perl's regular expressions resemble the DFAs (deterministic finite automata) of the `egrep(1)` program, they are in fact implemented as NFAs (non-deterministic finite automata) to allow backtracking and backreferencing. And they aren't POSIX-style either, because those guarantee worst-case behavior for all cases. (It seems that some people prefer guarantees of consistency, even when what's guaranteed is slowness.) See the book "Mastering Regular Expressions" (from O'Reilly) by Jeffrey Friedl for all the details you could ever hope to know on these matters (a full citation appears in [perlfaq2](#)).

What's wrong with using `grep` or `map` in a void context?

The problem is that both `grep` and `map` build a return list, regardless of the context. This means you're making Perl go to the trouble of building a list that you then just throw away. If the list is large, you waste both time and space. If your intent is to iterate over the list then use a `for` loop for this purpose.

How can I match strings with multibyte characters?

Starting from Perl 5.6 Perl has had some level of multibyte character support. Perl 5.8 or later is recommended. Supported multibyte character repertoires include Unicode, and legacy encodings through the Encode module. See [perluniintro](#), [perlunicode](#), and [Encode](#).

If you are stuck with older Perls, you can do Unicode with the `Unicode::String` module, and character conversions using the `Unicode::Map8` and `Unicode::Map` modules. If you are using Japanese encodings, you might try using the `jperl 5.005_03`.

Finally, the following set of approaches was offered by Jeffrey Friedl, whose article in issue #5 of The Perl Journal talks about this very matter.

Let's suppose you have some weird Martian encoding where pairs of ASCII uppercase letters encode single Martian letters (i.e. the two bytes "CV" make a single Martian letter, as do the two bytes "SG", "VS", "XX", etc.). Other bytes represent single characters, just like ASCII.

So, the string of Martian "I am CVSGXX!" uses 12 bytes to encode the nine characters 'I', ' ', 'a', 'm', ' ', 'C', 'V', 'S', 'G', 'X', 'X', '!'.
'CV', 'SG', 'XX', '!'.

Now, say you want to search for the single character `/GX/`. Perl doesn't know about Martian, so it'll find the two bytes "GX" in the "I am CVSGXX!" string, even though that character isn't there: it just looks like it is because "SG" is next to "XX", but there's no real "GX". This is a big problem.

Here are a few ways, all painful, to deal with it:

```
$martian =~ s/([A-Z][A-Z])/ $1 /g; # Make sure adjacent ``martian`` bytes
                                   # are no longer adjacent.
print "found GX!\n" if $martian =~ /GX/;
```

Or like this:

```
@chars = $martian =~ m/([A-Z][A-Z]|[^A-Z])/g;
# above is conceptually similar to: @chars = $text =~ m/(.)/g;
#
foreach $char (@chars) {
    print "found GX!\n", last if $char eq 'GX';
}
```

Or like this:

```
while ($martian =~ m/\G([A-Z][A-Z]|.)/gs) { # \G probably unneeded
    print "found GX!\n", last if $1 eq 'GX';
}
```

Or like this:

```
die "sorry, Perl doesn't (yet) have Martian support )-:\n";
```

There are many double- (and multi-) byte encodings commonly used these days. Some versions of these have 1-, 2-, 3-, and 4-byte characters, all mixed.

How do I match a pattern that is supplied by the user?

Well, if it's really a pattern, then just use

```
chomp($pattern = <STDIN>);
if ($line =~ /$pattern/) { }
```

Alternatively, since you have no guarantee that your user entered a valid regular expression, trap the exception this way:

```
if (eval { $line =~ /$pattern/ }) { }
```

If all you really want to search for a string, not a pattern, then you should either use the `index()` function,

which is made for string searching, or if you can't be disabused of using a pattern match on a non-pattern, then be sure to use `\Q...\E`, documented in [perlre](#).

```
$pattern = <STDIN>;

open (FILE, $input) or die "Couldn't open input $input: $!; aborting";
while (<FILE>) {
    print if /\Q$pattern\E/;
}
close FILE;
```

AUTHOR AND COPYRIGHT

Copyright (c) 1997–2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

NAME

perlfaq7 – General Perl Language Issues (\$Revision: 1.8 \$, \$Date: 2002/03/26 15:48:32 \$)

DESCRIPTION

This section deals with general Perl language issues that don't clearly fit into any of the other sections.

Can I get a BNF/yacc/RE for the Perl language?

There is no BNF, but you can paw your way through the yacc grammar in `perly.y` in the source distribution if you're particularly brave. The grammar relies on very smart tokenizing code, so be prepared to venture into `toke.c` as well.

In the words of Chaim Frenkel: "Perl's grammar can not be reduced to BNF. The work of parsing perl is distributed between yacc, the lexer, smoke and mirrors."

What are all these \$@%&* punctuation signs, and how do I know when to use them?

They are type specifiers, as detailed in [perldata](#):

```
$ for scalar values (number, string or reference)
@ for arrays
% for hashes (associative arrays)
& for subroutines (aka functions, procedures, methods)
* for all types of that symbol name. In version 4 you used them like
  pointers, but in modern perls you can just use references.
```

There are couple of other symbols that you're likely to encounter that aren't really type specifiers:

```
<> are used for inputting a record from a filehandle.
\  takes a reference to something.
```

Note that `<FILE` is *neither* the type specifier for files nor the name of the handle. It is the `< <` operator applied to the handle `FILE`. It reads one line (well, record—see [\\$/](#)) from the handle `FILE` in scalar context, or *all* lines in list context. When performing open, close, or any other operation besides `< <` on files, or even when talking about the handle, do *not* use the brackets. These are correct: `eof (FH)`, `seek (FH, 0, 2)` and "copying from STDIN to FILE".

Do I always/never have to quote my strings or use semicolons and commas?

Normally, a bareword doesn't need to be quoted, but in most cases probably should be (and must be under `use strict`). But a hash key consisting of a simple word (that isn't the name of a defined subroutine) and the left-hand operand to the `< =` operator both count as though they were quoted:

This	is like this
-----	-----
<code>\$foo{line}</code>	<code>\$foo{"line"}</code>
<code>bar => stuff</code>	<code>"bar" => stuff</code>

The final semicolon in a block is optional, as is the final comma in a list. Good style (see [perlstyle](#)) says to put them in except for one-liners:

```
if ($whoops) { exit 1 }
@nums = (1, 2, 3);

if ($whoops) {
    exit 1;
}
@lines = (
    "There Beren came from mountains cold",
    "And lost he wandered under leaves",
);
```

How do I skip some return values?

One way is to treat the return values as a list and index into it:

```
$dir = (getpwnam($user))[7];
```

Another way is to use `undef` as an element on the left-hand-side:

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

How do I temporarily block warnings?

If you are running Perl 5.6.0 or better, the `use warnings` pragma allows fine control of what warning are produced. See [perllexwarn](#) for more details.

```
{
    no warnings;           # temporarily turn off warnings
    $a = $b + $c;          # I know these might be undef
}
```

If you have an older version of Perl, the `$^W` variable (documented in [perlvar](#)) controls runtime warnings for a block:

```
{
    local $^W = 0;         # temporarily turn off warnings
    $a = $b + $c;          # I know these might be undef
}
```

Note that like all the punctuation variables, you cannot currently use `my()` on `$^W`, only `local()`.

What's an extension?

An extension is a way of calling compiled C code from Perl. Reading [perlxsut](#) is a good place to learn more about extensions.

Why do Perl operators have different precedence than C operators?

Actually, they don't. All C operators that Perl copies have the same precedence in Perl as they do in C. The problem is with operators that C doesn't have, especially functions that give a list context to everything on their right, eg. `print`, `chmod`, `exec`, and so on. Such functions are called "list operators" and appear as such in the precedence table in [perlop](#).

A common mistake is to write:

```
unlink $file || die "snafu";
```

This gets interpreted as:

```
unlink ($file || die "snafu");
```

To avoid this problem, either put in extra parentheses or use the super low precedence `or` operator:

```
(unlink $file) || die "snafu";
unlink $file or die "snafu";
```

The "English" operators (`and`, `or`, `xor`, and `not`) deliberately have precedence lower than that of list operators for just such situations as the one above.

Another operator with surprising precedence is exponentiation. It binds more tightly even than unary minus, making `-2**2` product a negative not a positive four. It is also right-associating, meaning that `2**3**2` is two raised to the ninth power, not eight squared.

Although it has the same precedence as in C, Perl's `?:` operator produces an lvalue. This assigns `$x` to either `$a` or `$b`, depending on the trueness of `$maybe`:

```
($maybe ? $a : $b) = $x;
```

How do I declare/create a structure?

In general, you don't "declare" a structure. Just use a (probably anonymous) hash reference. See [perlref](#) and [perlisc](#) for details. Here's an example:

```
$person = {};                                # new anonymous hash
$person->{AGE} = 24;                          # set field AGE to 24
$person->{NAME} = "Nat";                     # set field NAME to "Nat"
```

If you're looking for something a bit more rigorous, try [perltoot](#).

How do I create a module?

A module is a package that lives in a file of the same name. For example, the `Hello::There` module would live in `Hello/There.pm`. For details, read [perlmod](#). You'll also find [Exporter](#) helpful. If you're writing a C or mixed-language module with both C and Perl, then you should study [perlxs](#).

The `h2xs` program will create stubs for all the important stuff for you:

```
% h2xs -XA -n My::Module
```

The `-X` switch tells `h2xs` that you are not using XS extension code. The `-A` switch tells `h2xs` that you are not using the `AutoLoader`, and the `-n` switch specifies the name of the module. See [h2xs](#) for more details.

How do I create a class?

See [perltoot](#) for an introduction to classes and objects, as well as [perlobj](#) and [perlbot](#).

How can I tell if a variable is tainted?

You can use the `tainted()` function of the `Scalar::Util` module, available from CPAN (or included with Perl since release 5.8.0). See also [Laundering and Detecting Tainted Data in perlsec](#).

What's a closure?

Closures are documented in [perlref](#).

Closure is a computer science term with a precise but hard-to-explain meaning. Closures are implemented in Perl as anonymous subroutines with lasting references to lexical variables outside their own scopes. These lexicals magically refer to the variables that were around when the subroutine was defined (deep binding).

Closures make sense in any programming language where you can have the return value of a function be itself a function, as you can in Perl. Note that some languages provide anonymous functions but are not capable of providing proper closures: the Python language, for example. For more information on closures, check out any textbook on functional programming. Scheme is a language that not only supports but encourages closures.

Here's a classic function-generating function:

```
sub add_function_generator {
    return sub { shift + shift };
}

$add_sub = add_function_generator();
$sum = $add_sub->(4,5);           # $sum is 9 now.
```

The closure works as a *function template* with some customization slots left out to be filled later. The anonymous subroutine returned by `add_function_generator()` isn't technically a closure because it refers to no lexicals outside its own scope.

Contrast this with the following `make_adder()` function, in which the returned anonymous function contains a reference to a lexical variable outside the scope of that function itself. Such a reference requires that Perl return a proper closure, thus locking in for all time the value that the lexical had when the function was created.

```
sub make_adder {
    my $addpiece = shift;
```

```

        return sub { shift + $addpiece };
    }

    $f1 = make_adder(20);
    $f2 = make_adder(555);

```

Now `&$f1($n)` is always 20 plus whatever `$n` you pass in, whereas `&$f2($n)` is always 555 plus whatever `$n` you pass in. The `$addpiece` in the closure sticks around.

Closures are often used for less esoteric purposes. For example, when you want to pass in a bit of code into a function:

```

my $line;
timeout( 30, sub { $line = <STDIN> } );

```

If the code to execute had been passed in as a string, `< '$line = <STDIN>'`, there would have been no way for the hypothetical `timeout()` function to access the lexical variable `$line` back in its caller's scope.

What is variable suicide and how can I prevent it?

Variable suicide is when you (temporarily or permanently) lose the value of a variable. It is caused by scoping through `my()` and `local()` interacting with either closures or aliased `foreach()` iterator variables and subroutine arguments. It used to be easy to inadvertently lose a variable's value this way, but now it's much harder. Take this code:

```

my $f = "foo";
sub T {
    while ($i++ < 3) { my $f = $f; $f .= "bar"; print $f, "\n" }
}
T;
print "Finally $f\n";

```

The `$f` that has "bar" added to it three times should be a new `$f` (`my $f` should create a new local variable each time through the loop). It isn't, however. This was a bug, now fixed in the latest releases (tested against 5.004_05, 5.005_03, and 5.005_56).

How can I pass/return a {Function, FileHandle, Array, Hash, Method, Regex}?

With the exception of regexes, you need to pass references to these objects. See [Pass by Reference in perlsub](#) for this particular question, and [perlref](#) for information on references.

See "Passing Regexes", below, for information on passing regular expressions.

Passing Variables and Functions

Regular variables and functions are quite easy to pass: just pass in a reference to an existing or anonymous variable or function:

```

func( \$some_scalar );

func( \@some_array );
func( [ 1 .. 10 ] );

func( \%some_hash );
func( { this => 10, that => 20 } );

func( &some_func );
func( sub { $_[0] ** $_[1] } );

```

Passing Filehandles

To pass filehandles to subroutines, use the `*FH` or `*FH` notations. These are "typeglobs"—see [Typeglobs and Filehandles in perldata](#) and especially [Pass by Reference in perlsub](#) for more information.

Here's an excerpt:

If you're passing around filehandles, you could usually just use the bare typeglob, like `*STDOUT`, but typeglobs references would be better because they'll still work properly under use `strict 'refs'`. For example:

```
splutter(\*STDOUT);
sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

If you're planning on generating new filehandles, you could do this:

```
sub openit {
    my $path = shift;
    local *FH;
    return open (FH, $path) ? *FH : undef;
}
$fh = openit('< /etc/motd');
print <$fh>;
```

Passing Regexes

To pass regexes around, you'll need to be using a release of Perl sufficiently recent as to support the `qr//` construct, pass around strings and use an exception-trapping `eval`, or else be very, very clever.

Here's an example of how to pass in a string to be regex compared using `qr//`:

```
sub compare($$) {
    my ($val1, $regex) = @_;
    my $retval = $val1 =~ /$regex/;
    return $retval;
}
$match = compare("old McDonald", qr/d.*D/i);
```

Notice how `qr//` allows flags at the end. That pattern was compiled at compile time, although it was executed later. The nifty `qr//` notation wasn't introduced until the 5.005 release. Before that, you had to approach this problem much less intuitively. For example, here it is again if you don't have `qr//`:

```
sub compare($$) {
    my ($val1, $regex) = @_;
    my $retval = eval { $val1 =~ /$regex/ };
    die if $@;
    return $retval;
}
$match = compare("old McDonald", q/($?i)d.*D/);
```

Make sure you never say something like this:

```
return eval "\$val =~ /$regex/";    # WRONG
```

or someone can sneak shell escapes into the regex due to the double interpolation of the `eval` and the double-quoted string. For example:

```
$pattern_of_evil = 'danger ${ system("rm -rf * &") } danger';
eval "\$string =~ /$pattern_of_evil/";
```

Those preferring to be very, very clever might see the O'Reilly book, *Mastering Regular Expressions*, by Jeffrey Friedl. Page 273's `Build_MatchMany_Function()` is particularly interesting. A complete citation of this book is given in [perlfaq2](#).

Passing Methods

To pass an object method into a subroutine, you can do this:

```
call_a_lot(10, $some_obj, "methname")
sub call_a_lot {
    my ($count, $widget, $strick) = @_;
    for (my $i = 0; $i < $count; $i++) {
        $widget->$strick();
    }
}
```

Or, you can use a closure to bundle up the object, its method call, and arguments:

```
my $whatnot = sub { $some_obj->obfuscate(@args) };
func($whatnot);
sub func {
    my $code = shift;
    &$code();
}
```

You could also investigate the `can()` method in the `UNIVERSAL` class (part of the standard perl distribution).

How do I create a static variable?

As with most things in Perl, TMTOWTDI. What is a "static variable" in other languages could be either a function-private variable (visible only within a single function, retaining its value between calls to that function), or a file-private variable (visible only to functions within the file it was declared in) in Perl.

Here's code to implement a function-private variable:

```
BEGIN {
    my $counter = 42;
    sub prev_counter { return --$counter }
    sub next_counter { return $counter++ }
}
```

Now `prev_counter()` and `next_counter()` share a private variable `$counter` that was initialized at compile time.

To declare a file-private variable, you'll still use a `my()`, putting the declaration at the outer scope level at the top of the file. Assume this is in file `Pax.pm`:

```
package Pax;
my $started = scalar(localtime(time()));

sub begun { return $started }
```

When use `Pax` or `require Pax` loads this module, the variable will be initialized. It won't get garbage-collected the way most variables going out of scope do, because the `begun()` function cares about it, but no one else can get it. It is not called `$Pax::started` because its scope is unrelated to the package.

It's scoped to the file. You could conceivably have several packages in that same file all accessing the same private variable, but another file with the same package couldn't get to it.

See [Persistent Private Variables in perlsub](#) for details.

What's the difference between dynamic and lexical (static) scoping? Between `local()` and `my()`?

`local($x)` saves away the old value of the global variable `$x` and assigns a new value for the duration of the subroutine *which is visible in other functions called from that subroutine*. This is done at run-time, so is called dynamic scoping. `local()` always affects global variables, also called package variables or dynamic variables.

`my($x)` creates a new variable that is only visible in the current subroutine. This is done at compile-time, so it is called lexical or static scoping. `my()` always affects private variables, also called lexical variables or (improperly) static(ly scoped) variables.

For instance:

```
sub visible {
    print "var has value $var\n";
}

sub dynamic {
    local $var = 'local';    # new temporary value for the still-global
    visible();               #   variable called $var
}

sub lexical {
    my $var = 'private';    # new private variable, $var
    visible();              # (invisible outside of sub scope)
}

$var = 'global';

visible();                  # prints global
dynamic();                  # prints local
lexical();                  # prints global
```

Notice how at no point does the value "private" get printed. That's because `$var` only has that value within the block of the `lexical()` function, and it is hidden from called subroutine.

In summary, `local()` doesn't make what you think of as private, local variables. It gives a global variable a temporary value. `my()` is what you're looking for if you want private variables.

See *"Private Variables via `my()`"* and *"Temporary Values via `local()`"* for excruciating details.

How can I access a dynamic variable while a similarly named lexical is in scope?

You can do this via symbolic references, provided you haven't set `use strict "refs"`. So instead of `$var`, use `${'var'}`.

```
local $var = "global";
my    $var = "lexical";

print "lexical is $var\n";

no strict 'refs';
print "global  is ${'var'}\n";
```

If you know your package, you can just mention it explicitly, as in `$Some_Pack::var`. Note that the notation `$::var` is *not* the dynamic `$var` in the current package, but rather the one in the main package, as though you had written `$main::var`. Specifying the package directly makes you hard-code its name, but it executes faster and avoids running afoul of `use strict "refs"`.

What's the difference between deep and shallow binding?

In deep binding, lexical variables mentioned in anonymous subroutines are the same ones that were in scope when the subroutine was created. In shallow binding, they are whichever variables with the same names

happen to be in scope when the subroutine is called. Perl always uses deep binding of lexical variables (i.e., those created with `my()`). However, dynamic variables (aka global, local, or package variables) are effectively shallowly bound. Consider this just one more reason not to use them. See the answer to ["What's a closure?"](#).

Why doesn't "my(\$foo) = <FILE>" work right?

`my()` and `local()` give list context to the right hand side of `=`. The `<FH` read operation, like so many of Perl's functions and operators, can tell which context it was called in and behaves appropriately. In general, the `scalar()` function can help. This function does nothing to the data itself (contrary to popular myth) but rather tells its argument to behave in whatever its scalar fashion is. If that function doesn't have a defined scalar behavior, this of course doesn't help you (such as with `sort()`).

To enforce scalar context in this particular case, however, you need merely omit the parentheses:

```
local($foo) = <FILE>;           # WRONG
local($foo) = scalar(<FILE>);   # ok
local $foo = <FILE>;           # right
```

You should probably be using lexical variables anyway, although the issue is the same here:

```
my($foo) = <FILE>;             # WRONG
my $foo = <FILE>;              # right
```

How do I redefine a builtin function, operator, or method?

Why do you want to do that? :-)

If you want to override a predefined function, such as `open()`, then you'll have to import the new definition from a different module. See [Overriding Built-in Functions in perlsub](#). There's also an example in [Class::Template in perltoot](#).

If you want to overload a Perl operator, such as `+` or `**`, then you'll want to use the `use overload` pragma, documented in [overload](#).

If you're talking about obscuring method calls in parent classes, see [Overridden Methods in perltoot](#).

What's the difference between calling a function as `&foo` and `foo()`?

When you call a function as `&foo`, you allow that function access to your current `@_` values, and you bypass prototypes. The function doesn't get an empty `@_`—it gets yours! While not strictly speaking a bug (it's documented that way in [perlsub](#)), it would be hard to consider this a feature in most cases.

When you call your function as `&foo()`, then you *do* get a new `@_`, but prototyping is still circumvented.

Normally, you want to call a function using `foo()`. You may only omit the parentheses if the function is already known to the compiler because it already saw the definition (`use` but not `require`), or via a forward reference or `use subs` declaration. Even in this case, you get a clean `@_` without any of the old values leaking through where they don't belong.

How do I create a switch or case statement?

This is explained in more depth in the [perlsyn](#). Briefly, there's no official case statement, because of the variety of tests possible in Perl (numeric comparison, string comparison, glob comparison, regex matching, overloaded comparisons, ...). Larry couldn't decide how best to do this, so he left it out, even though it's been on the wish list since perl1.

Starting from Perl 5.8 to get switch and case one can use the Switch extension and say:

```
use Switch;
```

after which one has switch and case. It is not as fast as it could be because it's not really part of the language (it's done using source filters) but it is available, and it's very flexible.

But if one wants to use pure Perl, the general answer is to write a construct like this:

```

for ($variable_to_test) {
    if      (/pat1/) { }      # do something
    elsif  (/pat2/) { }      # do something else
    elsif  (/pat3/) { }      # do something else
    else    { }              # default
}

```

Here's a simple example of a switch based on pattern matching, this time lined up in a way to make it look more like a switch statement. We'll do a multiway conditional based on the type of reference stored in \$whatchamacallit:

```

SWITCH: for (ref $whatchamacallit) {
    /^$/          && die "not a reference";
    /SCALAR/      && do {
                        print_scalar($$ref);
                        last SWITCH;
                    };
    /ARRAY/       && do {
                        print_array(@$ref);
                        last SWITCH;
                    };
    /HASH/        && do {
                        print_hash(%$ref);
                        last SWITCH;
                    };
    /CODE/        && do {
                        warn "can't print function ref";
                        last SWITCH;
                    };

    # DEFAULT
    warn "User defined type skipped";
}

```

See `perlsyn/"Basic BLOCKs and Switch Statements"` for many other examples in this style.

Sometimes you should change the positions of the constant and the variable. For example, let's say you wanted to test which of many answers you were given, but in a case-insensitive way that also allows abbreviations. You can use the following technique if the strings all start with different characters or if you want to arrange the matches so that one takes precedence over another, as "SEND" has precedence over "STOP" here:

```

chomp($answer = <>);
if      ("SEND"  =~ /\Q$answer/i) { print "Action is send\n" }
elsif  ("STOP"  =~ /\Q$answer/i) { print "Action is stop\n" }
elsif  ("ABORT" =~ /\Q$answer/i) { print "Action is abort\n" }
elsif  ("LIST"  =~ /\Q$answer/i) { print "Action is list\n" }
elsif  ("EDIT"  =~ /\Q$answer/i) { print "Action is edit\n" }

```

A totally different approach is to create a hash of function references.

```

my %commands = (
    "happy" => \&joy,
    "sad",  => \&sullen,

```

```

        "done"  => sub { die "See ya!" },
        "mad"   => \&angry,
    );

    print "How are you? ";
    chomp($string = <STDIN>);
    if ($commands{$string}) {
        $commands{$string}->();
    } else {
        print "No such command: $string\n";
    }
}

```

How can I catch accesses to undefined variables/functions/methods?

The AUTOLOAD method, discussed in [Autoloading in perlsub](#) and [AUTOLOAD: Proxy Methods in perltoot](#), lets you capture calls to undefined functions and methods.

When it comes to undefined variables that would trigger a warning under `-w`, you can use a handler to trap the pseudo-signal `__WARN__` like this:

```

$SIG{__WARN__} = sub {
    for ( $_[0] ) {          # voici un switch statement
        /Use of uninitialized value/ && do {
            # promote warning to a fatal
            die $_;
        };
        # other warning cases to catch could go here;
        warn $_;
    }
};

```

Why can't a method included in this same file be found?

Some possible reasons: your inheritance is getting confused, you've misspelled the method name, or the object is of the wrong type. Check out [perltoot](#) for details about any of the above cases. You may also use `print ref($object)` to find out the class `$object` was blessed into.

Another possible reason for problems is because you've used the indirect object syntax (eg, `find Guru "Samy"`) on a class name before Perl has seen that such a package exists. It's wisest to make sure your packages are all defined before you start using them, which will be taken care of if you use the `use` statement instead of `require`. If not, make sure to use arrow notation (eg., `< Guru-find("Samy")`) instead. Object notation is explained in [perlobj](#).

Make sure to read about creating modules in [perlmod](#) and the perils of indirect objects in [Method Invocation in perlobj](#).

How can I find out my current package?

If you're just a random program, you can do this to find out what the currently compiled package is:

```
my $packname = __PACKAGE__;
```

But, if you're a method and you want to print an error message that includes the kind of object you were called on (which is not necessarily the same as the one in which you were compiled):

```

sub amethod {
    my $self = shift;
    my $class = ref($self) || $self;
    warn "called me from a $class object";
}

```

How can I comment out a large block of perl code?

You can use embedded POD to discard it. The `=for` directive lasts until the next paragraph (two consecutive newlines).

```
# program is here

=for nobody
This paragraph is commented out

# program continues
```

The `=begin` and `=end` directives can contain multiple paragraphs.

```
=begin comment text

all of this stuff

here will be ignored
by everyone

=end comment text
```

The pod directives cannot go just anywhere. You must put a pod directive where the parser is expecting a new statement, not just in the middle of an expression or some other arbitrary grammar production.

See [perlpod](#) for more details.

How do I clear a package?

Use this code, provided by Mark-Jason Dominus:

```
sub scrub_package {
    no strict 'refs';
    my $pack = shift;
    die "Shouldn't delete main package"
        if $pack eq "" || $pack eq "main";
    my $stash = *{$pack . '::'}{HASH};
    my $name;
    foreach $name (keys %$stash) {
        my $fullname = $pack . '::' . $name;
        # Get rid of everything with that name.
        undef $$fullname;
        undef @$fullname;
        undef %$fullname;
        undef &$fullname;
        undef *$fullname;
    }
}
```

Or, if you're using a recent release of Perl, you can just use the `Symbol::delete_package()` function instead.

How can I use a variable as a variable name?

Beginners often think they want to have a variable contain the name of a variable.

```
$fred    = 23;
$varname = "fred";
++$$varname;      # $fred now 24
```

This works *sometimes*, but it is a very bad idea for two reasons.

The first reason is that this technique *only works on global variables*. That means that if `$fred` is a lexical variable created with `my()` in the above example, the code wouldn't work at all: you'd accidentally access

the global and skip right over the private lexical altogether. Global variables are bad because they can easily collide accidentally and in general make for non-scalable and confusing code.

Symbolic references are forbidden under the `use strict` pragma. They are not true references and consequently are not reference counted or garbage collected.

The other reason why using a variable to hold the name of another variable is a bad idea is that the question often stems from a lack of understanding of Perl data structures, particularly hashes. By using symbolic references, you are just using the package's symbol-table hash (like `%main: :`) instead of a user-defined hash. The solution is to use your own hash or a real reference instead.

```
$fred      = 23;
$varname   = "fred";
%USER_VARS{$varname}++; # not $$varname++
```

There we're using the `%USER_VARS` hash instead of symbolic references. Sometimes this comes up in reading strings from the user with variable references and wanting to expand them to the values of your perl program's variables. This is also a bad idea because it conflates the program-addressable namespace and the user-addressable one. Instead of reading a string and expanding it to the actual contents of your program's own variables:

```
$str = 'this has a $fred and $barney in it';
$str =~ s/(\$(\w+))/${1}/eeg; # need double eval
```

it would be better to keep a hash around like `%USER_VARS` and have variable references actually refer to entries in that hash:

```
$str =~ s/\$(\w+)/$USER_VARS{$1}/g; # no /e here at all
```

That's faster, cleaner, and safer than the previous approach. Of course, you don't need to use a dollar sign. You could use your own scheme to make it less confusing, like bracketed percent symbols, etc.

```
$str = 'this has a %fred% and %barney% in it';
$str =~ s/%(\w+)/$USER_VARS{$1}/g; # no /e here at all
```

Another reason that folks sometimes think they want a variable to contain the name of a variable is because they don't know how to build proper data structures using hashes. For example, let's say they wanted two hashes in their program: `%fred` and `%barney`, and that they wanted to use another scalar variable to refer to those by name.

```
$name = "fred";
$$name{WIFE} = "wilma"; # set %fred

$name = "barney";
$$name{WIFE} = "betty"; # set %barney
```

This is still a symbolic reference, and is still saddled with the problems enumerated above. It would be far better to write:

```
$folks{"fred"}{WIFE} = "wilma";
$folks{"barney"}{WIFE} = "betty";
```

And just use a multilevel hash to start with.

The only times that you absolutely *must* use symbolic references are when you really must refer to the symbol table. This may be because it's something that can't take a real reference to, such as a format name. Doing so may also be important for method calls, since these always go through the symbol table for resolution.

In those cases, you would turn off `strict 'refs'` temporarily so you can play around with the symbol table. For example:

```
@colors = qw(red blue green yellow orange purple violet);
```

```
for my $name (@colors) {  
    no strict 'refs'; # renege for the block  
    *$name = sub { "<FONT COLOR=' $name' >@_</FONT>" };  
}
```

All those functions (`red()`, `blue()`, `green()`, etc.) appear to be separate, but the real code in the closure actually was compiled only once.

So, sometimes you might want to use symbolic references to directly manipulate the symbol table. This doesn't matter for formats, handles, and subroutines, because they are always global—you can't use `my()` on them. For scalars, arrays, and hashes, though—and usually for subroutines—you probably only want to use hard references.

AUTHOR AND COPYRIGHT

Copyright (c) 1997–2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

NAME

perlfaq8 – System Interaction (\$Revision: 1.8 \$, \$Date: 2002/05/16 12:41:42 \$)

DESCRIPTION

This section of the Perl FAQ covers questions involving operating system interaction. Topics include interprocess communication (IPC), control over the user–interface (keyboard, screen and pointing devices), and most anything else not related to data manipulation.

Read the FAQs and documentation specific to the port of perl to your operating system (eg, [perlvm](#)s, [perlplan9](#), ...). These should contain more detailed information on the vagaries of your perl.

How do I find out which operating system I'm running under?

The `$^O` variable (`$OSNAME` if you use English) contains an indication of the name of the operating system (not its release number) that your perl binary was built for.

How come `exec()` doesn't return?

Because that's what it does: it replaces your currently running program with a different one. If you want to keep going (as is probably the case if you're asking this question) use `system()` instead.

How do I do fancy stuff with the keyboard/screen/mouse?

How you access/control keyboards, screens, and pointing devices ("mice") is system–dependent. Try the following modules:

Keyboard

<code>Term::Cap</code>	Standard perl distribution
<code>Term::ReadKey</code>	CPAN
<code>Term::ReadLine::Gnu</code>	CPAN
<code>Term::ReadLine::Perl</code>	CPAN
<code>Term::Screen</code>	CPAN

Screen

<code>Term::Cap</code>	Standard perl distribution
<code>Curses</code>	CPAN
<code>Term::ANSIColor</code>	CPAN

Mouse

<code>Tk</code>	CPAN
-----------------	------

Some of these specific cases are shown below.

How do I print something out in color?

In general, you don't, because you don't know whether the recipient has a color–aware display device. If you know that they have an ANSI terminal that understands color, you can use the `Term::ANSIColor` module from CPAN:

```
use Term::ANSIColor;
print color("red"), "Stop!\n", color("reset");
print color("green"), "Go!\n", color("reset");
```

Or like this:

```
use Term::ANSIColor qw(:constants);
print RED, "Stop!\n", RESET;
print GREEN, "Go!\n", RESET;
```

How do I read just one key without waiting for a return key?

Controlling input buffering is a remarkably system–dependent matter. On many systems, you can just use the `stty` command as shown in [getc](#), but as you see, that's already getting you into portability snags.


```

open(TTY, "+</dev/tty") or die "no tty: $!";
system "stty cbreak </dev/tty >/dev/tty 2>&1";
$key = getc(TTY);    # perhaps this works
# OR ELSE
sysread(TTY, $key, 1#;probably this does
system "stty -cbreak </dev/tty >/dev/tty 2>&1";

```

The `Term::ReadKey` module from CPAN offers an easy-to-use interface that should be more efficient than shelling out to `stty` for each key. It even includes limited support for Windows.

```

use Term::ReadKey;
ReadMode('cbreak');
$key = ReadKey(0);
ReadMode('normal');

```

However, using the code requires that you have a working C compiler and can use it to build and install a CPAN module. Here's a solution using the standard POSIX module, which is already on your systems (assuming your system supports POSIX).

```

use HotKey;
$key = readkey();

```

And here's the `HotKey` module, which hides the somewhat mystifying calls to manipulate the POSIX `termios` structures.

```

# HotKey.pm
package HotKey;

@ISA = qw(Exporter);
@EXPORT = qw(cbreak cooked readkey);

use strict;
use POSIX qw(:termios_h);
my ($term, $oterm, $echo, $noecho, $fd_stdin);

$fd_stdin = fileno(STDIN);
$term      = POSIX::Termios->new();
$term->getattr($fd_stdin);
$oterm      = $term->getlflag();

$echo      = ECHO | ECHOK | ICANON;
$noecho    = $oterm & ~$echo;

sub cbreak {
    $term->setlflag($noecho); # ok, so i don't want echo either
    $term->setcc(VTIME, 1);
    $term->setattr($fd_stdin, TCSANOW);
}

sub cooked {
    $term->setlflag($oterm);
    $term->setcc(VTIME, 0);
    $term->setattr($fd_stdin, TCSANOW);
}

sub readkey {
    my $key = '';
    cbreak();
    sysread(STDIN, $key, 1);
    cooked();
    return $key;
}

```

```

    }
    END { cooked() }

    1;

```

How do I check whether input is ready on the keyboard?

The easiest way to do this is to read a key in nonblocking mode with the `Term::ReadKey` module from CPAN, passing it an argument of `-1` to indicate not to block:

```

use Term::ReadKey;

ReadMode('cbreak');

if (defined ($char = ReadKey(-1)) ) {
    # input was waiting and it was $char
} else {
    # no input was waiting
}

ReadMode('normal');           # restore normal tty settings

```

How do I clear the screen?

If you only have to do so infrequently, use `system`:

```
system("clear");
```

If you have to do this a lot, save the clear string so you can print it 100 times without calling a program 100 times:

```

$clear_string = `clear`;
print $clear_string;

```

If you're planning on doing other screen manipulations, like cursor positions, etc, you might wish to use `Term::Cap` module:

```

use Term::Cap;
$terminal = Term::Cap->Tgetent( {OSPEED => 9600} );
$clear_string = $terminal->Tputs('cl');

```

How do I get the screen size?

If you have `Term::ReadKey` module installed from CPAN, you can use it to fetch the width and height in characters and in pixels:

```

use Term::ReadKey;
($wchar, $hchar, $wpixels, $hpixels) = GetTerminalSize();

```

This is more portable than the raw `ioctl`, but not as illustrative:

```

require 'sys/ioctl.ph';
die "no TIOCGWINSZ " unless defined &TIOCGWINSZ;
open(TTY, "+</dev/tty") or die "No tty: $!";
unless (ioctl(TTY, &TIOCGWINSZ, $winsize='')) {
    die sprintf "$0: ioctl TIOCGWINSZ (%08x: $!)\n", &TIOCGWINSZ;
}
($row, $col, $xpixel, $ypixel) = unpack('S4', $winsize);
print "(row,col) = ($row,$col)";
print " (xpixel,ypixel) = ($xpixel,$ypixel)" if $xpixel || $ypixel;
print "\n";

```

How do I ask the user for a password?

(This question has nothing to do with the web. See a different FAQ for that.)

There's an example of this in [crypt](#). First, you put the terminal into "no echo" mode, then just read the password normally. You may do this with an old-style `ioctl()` function, POSIX terminal control (see [POSIX](#) or its documentation the Camel Book), or a call to the `stty` program, with varying degrees of portability.

You can also do this for most systems using the `Term::ReadKey` module from CPAN, which is easier to use and in theory more portable.

```
use Term::ReadKey;

ReadMode('noecho');
$password = ReadLine(0);
```

How do I read and write the serial port?

This depends on which operating system your program is running on. In the case of Unix, the serial ports will be accessible through files in `/dev`; on other systems, device names will doubtless differ. Several problem areas common to all device interaction are the following:

lockfiles

Your system may use lockfiles to control multiple access. Make sure you follow the correct protocol. Unpredictable behavior can result from multiple processes reading from one device.

open mode

If you expect to use both read and write operations on the device, you'll have to open it for update (see [open in perlfunc](#) for details). You may wish to open it without running the risk of blocking by using `sysopen()` and `O_RDWR|O_NDELAY|O_NOCTTY` from the `Fcntl` module (part of the standard perl distribution). See [sysopen in perlfunc](#) for more on this approach.

end of line

Some devices will be expecting a `"\r"` at the end of each line rather than a `"\n"`. In some ports of perl, `"\r"` and `"\n"` are different from their usual (Unix) ASCII values of `"\012"` and `"\015"`. You may have to give the numeric values you want directly, using octal (`"\015"`), hex (`"0x0D"`), or as a control-character specification (`"\cM"`).

```
print DEV "atv1\012";      # wrong, for some devices
print DEV "atv1\015";      # right, for some devices
```

Even though with normal text files a `"\n"` will do the trick, there is still no unified scheme for terminating a line that is portable between Unix, DOS/Win, and Macintosh, except to terminate *ALL* line ends with `"\015\012"`, and strip what you don't need from the output. This applies especially to socket I/O and autoflushing, discussed next.

flushing output

If you expect characters to get to your device when you `print()` them, you'll want to autoflush that filehandle. You can use `select()` and the `$|` variable to control autoflushing (see [\\$|](#) and [select](#), or [perlfaq5](#), "How do I flush/unbuffer an output filehandle? Why must I do this?"):

```
$oldh = select(DEV);
$| = 1;
select($oldh);
```

You'll also see code that does this without a temporary variable, as in

```
select((select(DEV), $| = 1)[0]);
```

Or if you don't mind pulling in a few thousand lines of code just because you're afraid of a little `$|` variable:

```
use IO::Handle;
DEV->autoflush(1);
```

As mentioned in the previous item, this still doesn't work when using socket I/O between Unix and Macintosh. You'll need to hard code your line terminators, in that case.

non-blocking input

If you are doing a blocking `read()` or `sysread()`, you'll have to arrange for an alarm handler to provide a timeout (see [alarm](#)). If you have a non-blocking open, you'll likely have a non-blocking read, which means you may have to use a 4-arg `select()` to determine whether I/O is ready on that device (see [select in perlfunc](#)).

While trying to read from his caller-id box, the notorious Jamie Zawinski <jwz@netscape.com, after much gnashing of teeth and fighting with `sysread`, `sysopen`, POSIX's `tcgetattr` business, and various other functions that go bump in the night, finally came up with this:

```
sub open_modem {
    use IPC::Open2;
    my $stty = `/bin/stty -g`;
    open2( \*MODEM_IN, \*MODEM_OUT, "cu -l$modem_device -s2400 2>&1");
    # starting cu hoses /dev/tty's stty settings, even when it has
    # been opened on a pipe...
    system("/bin/stty $stty");
    $_ = <MODEM_IN>;
    chomp;
    if ( !m/^Connected/ ) {
        print STDERR "$0: cu printed `$_' instead of `Connected'\n";
    }
}
```

How do I decode encrypted password files?

You spend lots and lots of money on dedicated hardware, but this is bound to get you talked about.

Seriously, you can't if they are Unix password files—the Unix password system employs one-way encryption. It's more like hashing than encryption. The best you can check is whether something else hashes to the same string. You can't turn a hash back into the original string. Programs like Crack can forcibly (and intelligently) try to guess passwords, but don't (can't) guarantee quick success.

If you're worried about users selecting bad passwords, you should proactively check when they try to change their password (by modifying `passwd(1)`, for example).

How do I start a process in the background?

You could use

```
system("cmd &")
```

or you could use `fork` as documented in [fork in perlfunc](#), with further examples in [perlipc](#). Some things to be aware of, if you're on a Unix-like system:

STDIN, STDOUT, and STDERR are shared

Both the main process and the backgrounded one (the "child" process) share the same STDIN, STDOUT and STDERR filehandles. If both try to access them at once, strange things can happen. You may want to close or reopen these for the child. You can get around this with opening a pipe (see [open in perlfunc](#)) but on some systems this means that the child process cannot outlive the parent.

Signals

You'll have to catch the SIGCHLD signal, and possibly SIGPIPE too. SIGCHLD is sent when the backgrounded process finishes. SIGPIPE is sent when you write to a filehandle whose child process has closed (an untrapped SIGPIPE can cause your program to silently die). This is not an issue with `system("cmd&")`.

Zombies

You have to be prepared to "reap" the child process when it finishes

```
$SIG{CHLD} = sub { wait };
```

See [Signals in perlipc](#) for other examples of code to do this. Zombies are not an issue with `system("prog &")`.

How do I trap control characters/signals?

You don't actually "trap" a control character. Instead, that character generates a signal which is sent to your terminal's currently foregrounded process group, which you then trap in your process. Signals are documented in [Signals in perlipc](#) and the section on "Signals" in the Camel.

Be warned that very few C libraries are re-entrant. Therefore, if you attempt to `print()` in a handler that got invoked during another stdio operation your internal structures will likely be in an inconsistent state, and your program will dump core. You can sometimes avoid this by using `syswrite()` instead of `print()`.

Unless you're exceedingly careful, the only safe things to do inside a signal handler are (1) set a variable and (2) exit. In the first case, you should only set a variable in such a way that `malloc()` is not called (eg, by setting a variable that already has a value).

For example:

```
$Interrupted = 0;    # to ensure it has a value
$SIG{INT} = sub {
    $Interrupted++;
    syswrite(STDERR, "ouch\n", 5);
}
```

However, because syscalls restart by default, you'll find that if you're in a "slow" call, such as `<FH`, `read()`, `connect()`, or `wait()`, that the only way to terminate them is by "longjumping" out; that is, by raising an exception. See the time-out handler for a blocking `flock()` in [Signals in perlipc](#) or the section on "Signals" in the Camel book.

How do I modify the shadow password file on a Unix system?

If perl was installed correctly and your shadow library was written properly, the `getpw*()` functions described in [perlfunc](#) should in theory provide (read-only) access to entries in the shadow password file. To change the file, make a new shadow password file (the format varies from system to system—see [passwd\(5\)](#) for specifics) and use `pwd_mkdb(8)` to install it (see [pwd_mkdb\(8\)](#) for more details).

How do I set the time and date?

Assuming you're running under sufficient permissions, you should be able to set the system-wide date and time by running the `date(1)` program. (There is no way to set the time and date on a per-process basis.) This mechanism will work for Unix, MS-DOS, Windows, and NT; the VMS equivalent is `set time`.

However, if all you want to do is change your time zone, you can probably get away with setting an environment variable:

```
$ENV{TZ} = "MST7MDT";          # unixish
$ENV{'SYS$TIMEZONE_DIFFERENTIAL'}="-5" # vms
system "trn comp.lang.perl.misc";
```

How can I `sleep()` or `alarm()` for under a second?

If you want finer granularity than the 1 second that the `sleep()` function provides, the easiest way is to use the `select()` function as documented in [select in perlfunc](#). Try the `Time::HiRes` and the `BSD::Itimer` modules (available from CPAN, and starting from Perl 5.8 `Time::HiRes` is part of the standard distribution).

How can I measure time under a second?

In general, you may not be able to. The `Time::HiRes` module (available from CPAN, and starting from Perl 5.8 part of the standard distribution) provides this functionality for some systems.

If your system supports both the `syscall()` function in Perl as well as a system call like `gettimeofday(2)`, then you may be able to do something like this:

```
require 'sys/syscall.ph';

$TIMEVAL_T = "LL";

$done = $start = pack($TIMEVAL_T, ());

syscall(&SYS_gettimeofday, $start, 0) != -1
    or die "gettimeofday: $!";

#####
# DO YOUR OPERATION HERE #
#####

syscall( &SYS_gettimeofday, $done, 0) != -1
    or die "gettimeofday: $!";

@start = unpack($TIMEVAL_T, $start);
@done  = unpack($TIMEVAL_T, $done);

# fix microseconds
for ($done[1], $start[1]) { $_ /= 1_000_000 }

$delta_time = sprintf "%.4f", ($done[0] + $done[1]
                                -
                                ($start[0] + $start[1]));
```

How can I do an `atexit()` or `setjmp()/longjmp()`? (Exception handling)

Release 5 of Perl added the `END` block, which can be used to simulate `atexit()`. Each package's `END` block is called when the program or thread ends (see [perlmod](#) manpage for more details).

For example, you can use this to make sure your filter program managed to finish its output without filling up the disk:

```
END {
    close(STDOUT) || die "stdout close failed: $!";
}
```

The `END` block isn't called when untrapped signals kill the program, though, so if you use `END` blocks you should also use

```
use sigtrap qw(die normal-signals);
```

Perl's exception-handling mechanism is its `eval()` operator. You can use `eval()` as `setjmp` and `die()` as `longjmp`. For details of this, see the section on signals, especially the time-out handler for a blocking `flock()` in [Signals in perlipc](#) or the section on "Signals" in the Camel Book.

If exception handling is all you're interested in, try the `exceptions.pl` library (part of the standard perl distribution).

If you want the `atexit()` syntax (and an `rmexit()` as well), try the `AtExit` module available from CPAN.

Why doesn't my sockets program work under System V (Solaris)? What does the error message "Protocol not supported" mean?

Some Sys-V based systems, notably Solaris 2.X, redefined some of the standard socket constants. Since these were constant across all architectures, they were often hardwired into perl code. The proper way to

deal with this is to "use Socket" to get the correct values.

Note that even though SunOS and Solaris are binary compatible, these values are different. Go figure.

How can I call my system's unique C functions from Perl?

In most cases, you write an external module to do it—see the answer to "Where can I learn about linking C with Perl? [h2xs, xsubpp]". However, if the function is a system call, and your system supports `syscall()`, you can use the `syscall` function (documented in [perlfunc](#)).

Remember to check the modules that came with your distribution, and CPAN as well—someone may already have written a module to do it.

Where do I get the include files to do `ioctl()` or `syscall()`?

Historically, these would be generated by the `h2ph` tool, part of the standard perl distribution. This program converts `cpp(1)` directives in C header files to files containing subroutine definitions, like `&SYS_getitimer`, which you can use as arguments to your functions. It doesn't work perfectly, but it usually gets most of the job done. Simple files like *errno.h*, *syscall.h*, and *socket.h* were fine, but the hard ones like *ioctl.h* nearly always need to hand-edited. Here's how to install the *.ph files:

1. become super-user
2. `cd /usr/include`
3. `h2ph *.h */*.h`

If your system supports dynamic loading, for reasons of portability and sanity you probably ought to use `h2xs` (also part of the standard perl distribution). This tool converts C header files to Perl extensions. See [perlxstut](#) for how to get started with `h2xs`.

If your system doesn't support dynamic loading, you still probably ought to use `h2xs`. See [perlxstut](#) and [ExtUtils::MakeMaker](#) for more information (in brief, just use **make perl** instead of a plain **make** to rebuild perl with a new static extension).

Why do `setuid` perl scripts complain about kernel problems?

Some operating systems have bugs in the kernel that make `setuid` scripts inherently insecure. Perl gives you a number of options (described in [perlsec](#)) to work around such systems.

How can I open a pipe both to and from a command?

The `IPC::Open2` module (part of the standard perl distribution) is an easy-to-use approach that internally uses `pipe()`, `fork()`, and `exec()` to do the job. Make sure you read the deadlock warnings in its documentation, though (see [IPC::Open2](#)). See [Bidirectional Communication with Another Process in *perlipc*](#) and [Bidirectional Communication with Yourself in *perlipc*](#)

You may also use the `IPC::Open3` module (part of the standard perl distribution), but be warned that it has a different order of arguments from `IPC::Open2` (see [IPC::Open3](#)).

Why can't I get the output of a command with `system()`?

You're confusing the purpose of `system()` and backticks (``). `system()` runs a command and returns exit status information (as a 16 bit value: the low 7 bits are the signal the process died from, if any, and the high 8 bits are the actual exit value). Backticks (``) run a command and return what it sent to `STDOUT`.

```
$exit_status = system("mail-users");
$output_string = `ls`;
```

How can I capture `STDERR` from an external command?

There are three basic ways of running external commands:

```
system $cmd;           # using system()
$output = ` $cmd `;     # using backticks (``)
open (PIPE, "cmd |");   # using open()
```

With `system()`, both `STDOUT` and `STDERR` will go the same place as the script's `STDOUT` and `STDERR`, unless the `system()` command redirects them. Backticks and `open()` read **only** the `STDOUT` of your command.

With any of these, you can change file descriptors before the call:

```
open(STDOUT, ">logfile");
system("ls");
```

or you can use Bourne shell file-descriptor redirection:

```
$output = `$cmd 2>some_file`;
open (PIPE, "cmd 2>some_file |");
```

You can also use file-descriptor redirection to make `STDERR` a duplicate of `STDOUT`:

```
$output = `$cmd 2>&1`;
open (PIPE, "cmd 2>&1 |");
```

Note that you *cannot* simply open `STDERR` to be a dup of `STDOUT` in your Perl program and avoid calling the shell to do the redirection. This doesn't work:

```
open(STDERR, ">&STDOUT");
$alloutput = `cmd args`; # stderr still escapes
```

This fails because the `open()` makes `STDERR` go to where `STDOUT` was going at the time of the `open()`. The backticks then make `STDOUT` go to a string, but don't change `STDERR` (which still goes to the old `STDOUT`).

Note that you *must* use Bourne shell (`sh(1)`) redirection syntax in backticks, not `csh(1)`! Details on why Perl's `system()` and backtick and pipe opens all use the Bourne shell are in the *versus/csh.whynot* article in the "Far More Than You Ever Wanted To Know" collection in <http://www.cpan.org/olddoc/FMTEYEWTK.tgz> . To capture a command's `STDERR` and `STDOUT` together:

```
$output = `cmd 2>&1`; # either with backticks
$pid = open(PH, "cmd 2>&1 |"); # or with an open pipe
while (<PH>) { } # plus a read
```

To capture a command's `STDOUT` but discard its `STDERR`:

```
$output = `cmd 2>/dev/null`; # either with backticks
$pid = open(PH, "cmd 2>/dev/null |"); # or with an open pipe
while (<PH>) { } # plus a read
```

To capture a command's `STDERR` but discard its `STDOUT`:

```
$output = `cmd 2>&1 1>/dev/null`; # either with backticks
$pid = open(PH, "cmd 2>&1 1>/dev/null |"); # or with an open pipe
while (<PH>) { } # plus a read
```

To exchange a command's `STDOUT` and `STDERR` in order to capture the `STDERR` but leave its `STDOUT` to come out our old `STDERR`:

```
$output = `cmd 3>&1 1>&2 2>&3 3>&-`; # either with backticks
$pid = open(PH, "cmd 3>&1 1>&2 2>&3 3>&-|"); # or with an open pipe
while (<PH>) { } # plus a read
```

To read both a command's `STDOUT` and its `STDERR` separately, it's easiest and safest to redirect them separately to files, and then read from those files when the program is done:

```
system("program args 1>/tmp/program.stdout 2>/tmp/program.stderr");
```

Ordering is important in all these examples. That's because the shell processes file descriptor redirections in

strictly left to right order.

```
system("prog args 1>tmpfile 2>&1");
system("prog args 2>&1 1>tmpfile");
```

The first command sends both standard out and standard error to the temporary file. The second command sends only the old standard output there, and the old standard error shows up on the old standard out.

Why doesn't `open()` return an error when a pipe open fails?

If the second argument to a piped `open` contains shell metacharacters, `perl fork()`s, then `exec()`s a shell to decode the metacharacters and eventually run the desired program. If the program couldn't be run, it's the shell that gets the message, not Perl. All your Perl program can find out is whether the shell itself could be successfully started. You can still capture the shell's `STDERR` and check it for error messages. See ["How can I capture `STDERR` from an external command?"](#) elsewhere in this document, or use the `IPC::Open3` module.

If there are no shell metacharacters in the argument of `open`, Perl runs the command directly, without using the shell, and can correctly report whether the command started.

What's wrong with using backticks in a void context?

Strictly speaking, nothing. Stylistically speaking, it's not a good way to write maintainable code. Perl has several operators for running external commands. Backticks are one; they collect the output from the command for use in your program. The `system` function is another; it doesn't do this.

Writing backticks in your program sends a clear message to the readers of your code that you wanted to collect the output of the command. Why send a clear message that isn't true?

Consider this line:

```
`cat /etc/termcap`;
```

You forgot to check `$?` to see whether the program even ran correctly. Even if you wrote

```
print `cat /etc/termcap`;
```

this code could and probably should be written as

```
system("cat /etc/termcap") == 0
    or die "cat program failed!";
```

which will get the output quickly (as it is generated, instead of only at the end) and also check the return value.

`system()` also provides direct control over whether shell wildcard processing may take place, whereas backticks do not.

How can I call backticks without shell processing?

This is a bit tricky. Instead of writing

```
@ok = `grep @opts '$search_string' @filenames`;
```

You have to do this:

```
my @ok = ();
if (open(GREP, "-|")) {
    while (<GREP>) {
        chomp;
        push(@ok, $_);
    }
    close GREP;
} else {
    exec 'grep', @opts, $search_string, @filenames;
}
```

Just as with `system()`, no shell escapes happen when you `exec()` a list. Further examples of this can be found in *Safe Pipe Opens in [perlipc](#)*.

Note that if you're stuck on Microsoft, no solution to this vexing issue is even possible. Even if Perl were to emulate `fork()`, you'd still be hosed, because Microsoft gives no `argc/argv`-style API. Their API always reparses from a single string, which is fundamentally wrong, but you're not likely to get the Gods of Redmond to acknowledge this and fix it for you.

Why can't my script read from STDIN after I gave it EOF (^D on Unix, ^Z on MS-DOS)?

Some `stdio`'s set error and eof flags that need clearing. The POSIX module defines `clearerr()` that you can use. That is the technically correct way to do it. Here are some less reliable workarounds:

- 1 Try keeping around the seekpointer and go there, like this:


```
$where = tell(LOG);
seek(LOG, $where, 0);
```
- 2 If that doesn't work, try seeking to a different part of the file and then back.
- 3 If that doesn't work, try seeking to a different part of the file, reading something, and then seeking back.
- 4 If that doesn't work, give up on your `stdio` package and use `sysread`.

How can I convert my shell script to perl?

Learn Perl and rewrite it. Seriously, there's no simple converter. Things that are awkward to do in the shell are easy to do in Perl, and this very awkwardness is what would make a shell-perl converter nigh-on impossible to write. By rewriting it, you'll think about what you're really trying to do, and hopefully will escape the shell's pipeline datastream paradigm, which while convenient for some matters, causes many inefficiencies.

Can I use perl to run a telnet or ftp session?

Try the `Net::FTP`, `TCP::Client`, and `Net::Telnet` modules (available from CPAN). <http://www.cpan.org/scripts/netstuff/telnet.emul.shar> will also help for emulating the telnet protocol, but `Net::Telnet` is quite probably easier to use..

If all you want to do is pretend to be telnet but don't need the initial telnet handshaking, then the standard dual-process approach will suffice:

```
use IO::Socket;                # new in 5.004
$handle = IO::Socket::INET->new('www.perl.com:80')
    || die "can't connect to port 80 on www.perl.com: $!";
$handle->autoflush(1);
if (fork()) {                  # XXX: undef means failure
    select($handle);
    print while <STDIN>;       # everything from stdin to socket
} else {
    print while <$handle>;     # everything from socket to stdout
}
close $handle;
exit;
```

How can I write expect in Perl?

Once upon a time, there was a library called `chat2.pl` (part of the standard perl distribution), which never really got finished. If you find it somewhere, *don't use it*. These days, your best bet is to look at the `Expect` module available from CPAN, which also requires two other modules from CPAN, `IO::Pty` and `IO::Stty`.

Is there a way to hide perl's command line from programs such as "ps"?

First of all note that if you're doing this for security reasons (to avoid people seeing passwords, for example) then you should rewrite your program so that critical information is never given as an argument. Hiding the

arguments won't make your program completely secure.

To actually alter the visible command line, you can assign to the variable `$0` as documented in [perlvar](#). This won't work on all operating systems, though. Daemon programs like sendmail place their state there, as in:

```
$0 = "orcus [accepting connections]";
```

I {changed directory, modified my environment} in a perl script. How come the change disappeared when I exited the script? How do I get my changes to be visible?

Unix

In the strictest sense, it can't be done—the script executes as a different process from the shell it was started from. Changes to a process are not reflected in its parent—only in any children created after the change. There is shell magic that may allow you to fake it by `eval()`ing the script's output in your shell; check out the `comp.unix.questions` FAQ for details.

How do I close a process's filehandle without waiting for it to complete?

Assuming your system supports such things, just send an appropriate signal to the process (see [kill in perlfunc](#)). It's common to first send a `TERM` signal, wait a little bit, and then send a `KILL` signal to finish it off.

How do I fork a daemon process?

If by daemon process you mean one that's detached (disassociated from its tty), then the following process is reported to work on most Unixish systems. Non-Unix users should check their `Your_OS::Process` module for other solutions.

- Open `/dev/tty` and use the `TIOCNOTTY` ioctl on it. See [tty\(4\)](#) for details. Or better yet, you can just use the `POSIX::setsid()` function, so you don't have to worry about process groups.
- Change directory to `/`
- Reopen `STDIN`, `STDOUT`, and `STDERR` so they're not connected to the old tty.
- Background yourself like this:

```
fork && exit;
```

The `Proc::Daemon` module, available from CPAN, provides a function to perform these actions for you.

How do I find out if I'm running interactively or not?

Good question. Sometimes `-t STDIN` and `-t STDOUT` can give clues, sometimes not.

```
if (-t STDIN && -t STDOUT) {
    print "Now what? ";
}
```

On POSIX systems, you can test whether your own process group matches the current process group of your controlling terminal as follows:

```
use POSIX qw/getpgrp tcgetpgrp/;
open(TTY, "/dev/tty") or die $!;
$tpgrp = tcgetpgrp(fileno(*TTY));
$pgrp = getpgrp();
if ($tpgrp == $pgrp) {
    print "foreground\n";
} else {
    print "background\n";
}
```

How do I timeout a slow event?

Use the `alarm()` function, probably in conjunction with a signal handler, as documented in [Signals in perlipc](#) and the section on “Signals” in the Camel. You may instead use the more flexible

Sys::AlarmCall module available from CPAN.

How do I set CPU limits?

Use the BSD::Resource module from CPAN.

How do I avoid zombies on a Unix system?

Use the reaper code from *Signals in perlipc* to call `wait()` when a SIGCHLD is received, or else use the double-fork technique described in *fork*.

How do I use an SQL database?

There are a number of excellent interfaces to SQL databases. See the DBD::* modules available from <http://www.cpan.org/modules/DBD>. A lot of information on this can be found at <http://dbi.perl.org/>

How do I make a `system()` exit on control-C?

You can't. You need to imitate the `system()` call (see *perlipc* for sample code) and then have a signal handler for the INT signal that passes the signal on to the subprocess. Or you can check for it:

```
$rc = system($cmd);
if ($rc & 127) { die "signal death" }
```

How do I open a file without blocking?

If you're lucky enough to be using a system that supports non-blocking reads (most Unixish systems do), you need only to use the O_NDELAY or O_NONBLOCK flag from the Fcntl module in conjunction with `sysopen()`:

```
use Fcntl;
sysopen(FH, "/tmp/somefile", O_WRONLY|O_NDELAY|O_CREAT, 0644)
    or die "can't open /tmp/somefile: $!";
```

How do I install a module from CPAN?

The easiest way is to have a module also named CPAN do it for you. This module comes with perl version 5.004 and later.

```
$ perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.59_54)
ReadLine support enabled

cpan> install Some::Module
```

To manually install the CPAN module, or any well-behaved CPAN module for that matter, follow these steps:

- 1 Unpack the source into a temporary area.
- 2


```
perl Makefile.PL
```
- 3


```
make
```
- 4


```
make test
```
- 5


```
make install
```

If your version of perl is compiled without dynamic loading, then you just need to replace step 3 (**make**) with **make perl** and you will get a new *perl* binary with your extension linked in.

See *ExtUtils::MakeMaker* for more details on building extensions. See also the next question, “What’s the difference between require and use?”.

What's the difference between `require` and `use`?

Perl offers several different ways to include code from one file into another. Here are the deltas between the various inclusion constructs:

- 1) `do $file` is like `eval 'cat $file'`, except the former
 - 1.1: searches `@INC` and updates `%INC`.
 - 1.2: bequeaths an **unrelated** lexical scope on the eval'ed code.
- 2) `require $file` is like `do $file`, except the former
 - 2.1: checks for redundant loading, skipping already loaded files.
 - 2.2: raises an exception on failure to find, compile, or execute `$file`.
- 3) `require Module` is like `require "Module.pm"`, except the former
 - 3.1: translates each `::` into your system's directory separator.
 - 3.2: primes the parser to disambiguate class `Module` as an indirect object.
- 4) `use Module` is like `require Module`, except the former
 - 4.1: loads the module at compile time, not run-time.
 - 4.2: imports symbols and semantics from that package to the current one.

In general, you usually want `use` and a proper Perl module.

How do I keep my own module/library directory?

When you build modules, use the `PREFIX` option when generating Makefiles:

```
perl Makefile.PL PREFIX=/u/mydir/perl
```

then either set the `PERL5LIB` environment variable before you run scripts that use the modules/libraries (see [perlrun](#)) or say

```
use lib '/u/mydir/perl';
```

This is almost the same as

```
BEGIN {
    unshift(@INC, '/u/mydir/perl');
}
```

except that the `lib` module checks for machine-dependent subdirectories. See Perl's [lib](#) for more information.

How do I add the directory my program lives in to the module/library search path?

```
use FindBin;
use lib "$FindBin::Bin";
use your_own_modules;
```

How do I add a directory to my include path at runtime?

Here are the suggested ways of modifying your include path:

```
the PERLLIB environment variable
the PERL5LIB environment variable
the perl -Idir command line flag
the use lib pragma, as in
    use lib "$ENV{HOME}/myown_perllib";
```

The latter is particularly useful because it knows about machine dependent architectures. The `lib.pm` pragmatic module was first included with the 5.002 release of Perl.

What is `socket.ph` and where do I get it?

It's a perl4-style file defining values for system networking constants. Sometimes it is built using `h2ph` when Perl is installed, but other times it is not. Modern programs use `Socket`; instead.

AUTHOR AND COPYRIGHT

Copyright (c) 1997–2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

NAME

perlfaq9 – Networking (\$Revision: 1.9 \$, \$Date: 2002/04/07 18:46:13 \$)

DESCRIPTION

This section deals with questions related to networking, the internet, and a few on the web.

What is the correct form of response from a CGI script?

(Alan Flavell <flavell+www@a5.ph.gla.ac.uk answers...)

The Common Gateway Interface (CGI) specifies a software interface between a program ("CGI script") and a web server (HTTPD). It is not specific to Perl, and has its own FAQs and tutorials, and usenet group, comp.infosystems.www.authoring.cgi

The original CGI specification is at: <http://hoohoo.ncsa.uiuc.edu/cgi/>

Current best-practice RFC draft at: <http://CGI-Spec.Golux.Com/>

Other relevant documentation listed in: http://www.perl.org/CGI_MetaFAQ.html

These Perl FAQs very selectively cover some CGI issues. However, Perl programmers are strongly advised to use the CGI.pm module, to take care of the details for them.

The similarity between CGI response headers (defined in the CGI specification) and HTTP response headers (defined in the HTTP specification, RFC2616) is intentional, but can sometimes be confusing.

The CGI specification defines two kinds of script: the "Parsed Header" script, and the "Non Parsed Header" (NPH) script. Check your server documentation to see what it supports. "Parsed Header" scripts are simpler in various respects. The CGI specification allows any of the usual newline representations in the CGI response (it's the server's job to create an accurate HTTP response based on it). So "\n" written in text mode is technically correct, and recommended. NPH scripts are more tricky: they must put out a complete and accurate set of HTTP transaction response headers; the HTTP specification calls for records to be terminated with carriage-return and line-feed, i.e ASCII \015\012 written in binary mode.

Using CGI.pm gives excellent platform independence, including EBCDIC systems. CGI.pm selects an appropriate newline representation (\$CGI : : CRLF) and sets binmode as appropriate.

My CGI script runs from the command line but not the browser. (500 Server Error)

Several things could be wrong. You can go through the "Troubleshooting Perl CGI scripts" guide at

http://www.perl.org/troubleshooting_CGI.html

If, after that, you can demonstrate that you've read the FAQs and that your problem isn't something simple that can be easily answered, you'll probably receive a courteous and useful reply to your question if you post it on comp.infosystems.www.authoring.cgi (if it's something to do with HTTP or the CGI protocols). Questions that appear to be Perl questions but are really CGI ones that are posted to comp.lang.perl.misc are not so well received.

The useful FAQs, related documents, and troubleshooting guides are listed in the CGI Meta FAQ:

http://www.perl.org/CGI_MetaFAQ.html

How can I get better error messages from a CGI program?

Use the CGI::Carp module. It replaces warn and die, plus the normal Carp modules carp, croak, and confess functions with more verbose and safer versions. It still sends them to the normal server error log.

```
use CGI::Carp;
warn "This is a complaint";
die "But this one is serious";
```

The following use of CGI::Carp also redirects errors to a file of your choice, placed in a BEGIN block to catch compile-time warnings as well:

```
BEGIN {
    use CGI::Carp qw(carpout);
    open(LOG, ">>/var/local/cgi-logs/mycgi-log")
        or die "Unable to append to mycgi-log: $!\n";
    carpout(*LOG);
}
```

You can even arrange for fatal errors to go back to the client browser, which is nice for your own debugging, but might confuse the end user.

```
use CGI::Carp qw(fatalsToBrowser);
die "Bad error here";
```

Even if the error happens before you get the HTTP header out, the module will try to take care of this to avoid the dreaded server 500 errors. Normal warnings still go out to the server error log (or wherever you've sent them with `carpout`) with the application name and date stamp prepended.

How do I remove HTML from a string?

The most correct way (albeit not the fastest) is to use `HTML::Parser` from CPAN. Another mostly correct way is to use `HTML::FormatText` which not only removes HTML but also attempts to do a little simple formatting of the resulting plain text.

Many folks attempt a simple-minded regular expression approach, like `< s/<.*?//g`, but that fails in many cases because the tags may continue over line breaks, they may contain quoted angle-brackets, or HTML comment may be present. Plus, folks forget to convert entities—like `<`; for example.

Here's one "simple-minded" approach, that works for most files:

```
#!/usr/bin/perl -p0777
s/<(?: [^>'"]* | (['"] ) . * ? \1 ) * > / /gs
```

If you want a more complete solution, see the 3-stage `striphtml` program in http://www.cpan.org/authors/Tom_Christiansen/scripts/striphtml.gz.

Here are some tricky cases that you should think about when picking a solution:

```
<IMG SRC = "foo.gif" ALT = "A > B">

<IMG SRC = "foo.gif"
    ALT = "A > B">

<!-- <A comment> -->

<script>if (a<b && a>c)</script>

<# Just data #>

<![INCLUDE CDATA [ >>>>>>>>>> ]]>
```

If HTML comments include other tags, those solutions would also break on text like this:

```
<!-- This section commented out.
    <B>You can't see me!</B>
-->
```

How do I extract URLs?

You can easily extract all sorts of URLs from HTML with `HTML::SimpleLinkExtor` which handles anchors, images, objects, frames, and many other tags that can contain a URL. If you need anything more complex, you can create your own subclass of `HTML::LinkExtor` or `HTML::Parser`. You might even use `HTML::SimpleLinkExtor` as an example for something specifically suited to your needs.

Less complete solutions involving regular expressions can save you a lot of processing time if you know that the input is simple. One solution from Tom Christiansen runs 100 times faster than most module based approaches but only extracts URLs from anchors where the first attribute is `HREF` and there are no other

attributes.

```
#!/usr/bin/perl -n00
# qxurl - tchrist@perl.com
print "$2\n" while m{
    < \s*
        A \s+ HREF \s* = \s* (["']) (.*?) \1
    \s* >
}gsix;
```

How do I download a file from the user's machine? How do I open a file on another machine?

In the context of an HTML form, you can use what's known as **multipart/form-data** encoding. The CGI.pm module (available from CPAN) supports this in the `start_multipart_form()` method, which isn't the same as the `startform()` method.

How do I make a pop-up menu in HTML?

Use the `<SELECT` and `<OPTION` tags. The CGI.pm module (available from CPAN) supports this widget, as well as many others, including some that it cleverly synthesizes on its own.

How do I fetch an HTML file?

One approach, if you have the lynx text-based HTML browser installed on your system, is this:

```
$html_code = `lynx -source $url`;
$text_data = `lynx -dump $url`;
```

The libwww-perl (LWP) modules from CPAN provide a more powerful way to do this. They don't require lynx, but like lynx, can still work through proxies:

```
# simplest version
use LWP::Simple;
$content = get($URL);

# or print HTML from a URL
use LWP::Simple;
getprint "http://www.linpro.no/lwp/";

# or print ASCII from HTML from a URL
# also need HTML-Tree package from CPAN
use LWP::Simple;
use HTML::Parser;
use HTML::FormatText;
my ($html, $ascii);
$html = get("http://www.perl.com/");
defined $html
    or die "Can't fetch HTML from http://www.perl.com/";
$ascii = HTML::FormatText->new->format(parse_html($html));
print $ascii;
```

How do I automate an HTML form submission?

If you're submitting values using the GET method, create a URL and encode the form using the `query_form` method:

```
use LWP::Simple;
use URI::URL;

my $url = url('http://www.perl.com/cgi-bin/cpan_mod');
$url->query_form(module => 'DB_File', readme => 1);
$content = get($url);
```

If you're using the POST method, create your own user agent and encode the content appropriately.

```

use HTTP::Request::Common qw(POST);
use LWP::UserAgent;

$sua = LWP::UserAgent->new();
my $req = POST 'http://www.perl.com/cgi-bin/cpan_mod',
               [ module => 'DB_File', readme => 1 ];
$content = $sua->request($req)->as_string;

```

How do I decode or create those %-encodings on the web?

If you are writing a CGI script, you should be using the CGI.pm module that comes with perl, or some other equivalent module. The CGI module automatically decodes queries for you, and provides an `escape()` function to handle encoding.

The best source of detailed information on URI encoding is RFC 2396. Basically, the following substitutions do it:

```

s/([^\w()'*~!.-])/sprintf '%02x', ord $1/eg;    # encode
s/%([A-Za-f\d]{2})/chr hex $1/eg;              # decode

```

However, you should only apply them to individual URI components, not the entire URI, otherwise you'll lose information and generally mess things up. If that didn't explain it, don't worry. Just go read section 2 of the RFC, it's probably the best explanation there is.

RFC 2396 also contains a lot of other useful information, including a regexp for breaking any arbitrary URI into components (Appendix B).

How do I redirect to another page?

Specify the complete URL of the destination (even if it is on the same server). This is one of the two different kinds of CGI "Location:" responses which are defined in the CGI specification for a Parsed Headers script. The other kind (an absolute URLpath) is resolved internally to the server without any HTTP redirection. The CGI specifications do not allow relative URLs in either case.

Use of CGI.pm is strongly recommended. This example shows redirection with a complete URL. This redirection is handled by the web browser.

```

use CGI qw/:standard/;

my $url = 'http://www.cpan.org/';
print redirect($url);

```

This example shows a redirection with an absolute URLpath. This redirection is handled by the local web server.

```

my $url = '/CPAN/index.html';
print redirect($url);

```

But if coded directly, it could be as follows (the final "\n" is shown separately, for clarity), using either a complete URL or an absolute URLpath.

```

print "Location: $url\n";    # CGI response header
print "\n";                  # end of headers

```

How do I put a password on my web pages?

That depends. You'll need to read the documentation for your web server, or perhaps check some of the other FAQs referenced above.

How do I edit my .htpasswd and .htgroup files with Perl?

The HTTPD::UserAdmin and HTTPD::GroupAdmin modules provide a consistent OO interface to these files, regardless of how they're stored. Databases may be text, dbm, Berkeley DB or any database with a DBI compatible driver. HTTPD::UserAdmin supports files used by the 'Basic' and 'Digest' authentication schemes. Here's an example:

```
use HTTPD::UserAdmin ();
HTTPD::UserAdmin
    ->new(DB => "/foo/.htpasswd")
    ->add($username => $password);
```

How do I make sure users can't enter values into a form that cause my CGI script to do bad things?

See the security references listed in the CGI Meta FAQ

http://www.perl.org/CGI_MetaFAQ.html

How do I parse a mail header?

For a quick-and-dirty solution, try this solution derived from *split*:

```
$/ = '';
$header = <MSG>;
$header =~ s/\n\s+/ /g;      # merge continuation lines
%head = ( UNIX_FROM_LINE, split /^([\w]+):\s*/m, $header );
```

That solution doesn't do well if, for example, you're trying to maintain all the Received lines. A more complete approach is to use the Mail::Header module from CPAN (part of the MailTools package).

How do I decode a CGI form?

You use a standard module, probably CGI.pm. Under no circumstances should you attempt to do so by hand!

You'll see a lot of CGI programs that blindly read from STDIN the number of bytes equal to CONTENT_LENGTH for POSTs, or grab QUERY_STRING for decoding GETs. These programs are very poorly written. They only work sometimes. They typically forget to check the return value of the `read()` system call, which is a cardinal sin. They don't handle HEAD requests. They don't handle multipart forms used for file uploads. They don't deal with GET/POST combinations where query fields are in more than one place. They don't deal with keywords in the query string.

In short, they're bad hacks. Resist them at all costs. Please do not be tempted to reinvent the wheel. Instead, use the CGI.pm or CGI_Lite.pm (available from CPAN), or if you're trapped in the module-free land of perl1 .. perl4, you might look into cgi-lib.pl (available from <http://cgi-lib.stanford.edu/cgi-lib/>).

Make sure you know whether to use a GET or a POST in your form. GETs should only be used for something that doesn't update the server. Otherwise you can get mangled databases and repeated feedback mail messages. The fancy word for this is "idempotency". This simply means that there should be no difference between making a GET request for a particular URL once or multiple times. This is because the HTTP protocol definition says that a GET request may be cached by the browser, or server, or an intervening proxy. POST requests cannot be cached, because each request is independent and matters. Typically, POST requests change or depend on state on the server (query or update a database, send mail, or purchase a computer).

How do I check a valid mail address?

You can't, at least, not in real time. Bummer, eh?

Without sending mail to the address and seeing whether there's a human on the other hand to answer you, you cannot determine whether a mail address is valid. Even if you apply the mail header standard, you can have problems, because there are deliverable addresses that aren't RFC-822 (the mail header standard) compliant, and addresses that aren't deliverable which are compliant.

Many are tempted to try to eliminate many frequently-invalid mail addresses with a simple regex, such as `/^[\w.-]+@(?:[\w-]+\.)+\w+$/`. It's a very bad idea. However, this also throws out many valid ones, and says nothing about potential deliverability, so it is not suggested. Instead, see http://www.cpan.org/authors/Tom_Christiansen/scripts/ckaddr.gz, which actually checks against the full RFC spec (except for nested comments), looks for addresses you may not wish to accept mail to (say, Bill Clinton or your postmaster), and then makes sure that the hostname given can be looked up in the DNS MX

records. It's not fast, but it works for what it tries to do.

Our best advice for verifying a person's mail address is to have them enter their address twice, just as you normally do to change a password. This usually weeds out typos. If both versions match, send mail to that address with a personal message that looks somewhat like:

```
Dear someuser@host.com,

Please confirm the mail address you gave us Wed May 6 09:38:41
MDT 1998 by replying to this message. Include the string
"Rumpelstiltskin" in that reply, but spelled in reverse; that is,
start with "Nik...". Once this is done, your confirmed address will
be entered into our records.
```

If you get the message back and they've followed your directions, you can be reasonably assured that it's real.

A related strategy that's less open to forgery is to give them a PIN (personal ID number). Record the address and PIN (best that it be a random one) for later processing. In the mail you send, ask them to include the PIN in their reply. But if it bounces, or the message is included via a "vacation" script, it'll be there anyway. So it's best to ask them to mail back a slight alteration of the PIN, such as with the characters reversed, one added or subtracted to each digit, etc.

How do I decode a MIME/BASE64 string?

The MIME-Base64 package (available from CPAN) handles this as well as the MIME/QP encoding. Decoding BASE64 becomes as simple as:

```
use MIME::Base64;
$decoded = decode_base64($encoded);
```

The MIME-Tools package (available from CPAN) supports extraction with decoding of BASE64 encoded attachments and content directly from email messages.

If the string to decode is short (less than 84 bytes long) a more direct approach is to use the `unpack()` function's "u" format after minor transliterations:

```
tr#A-Za-z0-9+/##cd;           # remove non-base64 chars
tr#A-Za-z0-9+/# -_#;          # convert to uuencoded format
$len = pack("c", 32 + 0.75*length); # compute length byte
print unpack("u", $len . $_);   # uudecode and print
```

How do I return the user's mail address?

On systems that support `getpwuid`, the `$<` variable, and the `Sys::Hostname` module (which is part of the standard perl distribution), you can probably try using something like this:

```
use Sys::Hostname;
$address = sprintf('%s@%s', scalar getpwuid($<), hostname);
```

Company policies on mail address can mean that this generates addresses that the company's mail system will not accept, so you should ask for users' mail addresses when this matters. Furthermore, not all systems on which Perl runs are so forthcoming with this information as is Unix.

The `Mail::Util` module from CPAN (part of the MailTools package) provides a `mailaddress()` function that tries to guess the mail address of the user. It makes a more intelligent guess than the code above, using information given when the module was installed, but it could still be incorrect. Again, the best way is often just to ask the user.

How do I send mail?

Use the `sendmail` program directly:

```
open(SENDMAIL, "|/usr/lib/sendmail -oi -t -odq")
    or die "Can't fork for sendmail: $!\n";
```

```

print SENDMAIL <<"EOF";
From: User Originating Mail <me\@host>
To: Final Destination <you\@otherhost>
Subject: A relevant subject line

Body of the message goes here after the blank line
in as many lines as you like.
EOF
close(SENDMAIL)      or warn "sendmail didn't close nicely";

```

The **-oi** option prevents sendmail from interpreting a line consisting of a single dot as "end of message". The **-t** option says to use the headers to decide who to send the message to, and **-odq** says to put the message into the queue. This last option means your message won't be immediately delivered, so leave it out if you want immediate delivery.

Alternate, less convenient approaches include calling mail (sometimes called mailx) directly or simply opening up port 25 have having an intimate conversation between just you and the remote SMTP daemon, probably sendmail.

Or you might be able use the CPAN module Mail::Mailer:

```

use Mail::Mailer;

$mailer = Mail::Mailer->new();
$mailer->open({ From    => $from_address,
                 To      => $to_address,
                 Subject => $subject,
                 })
    or die "Can't open: $!\n";
print $mailer $body;
$mailer->close();

```

The Mail::Internet module uses Net::SMTP which is less Unix-centric than Mail::Mailer, but less reliable. Avoid raw SMTP commands. There are many reasons to use a mail transport agent like sendmail. These include queuing, MX records, and security.

How do I use MIME to make an attachment to a mail message?

This answer is extracted directly from the MIME::Lite documentation. Create a multipart message (i.e., one with attachments).

```

use MIME::Lite;

### Create a new multipart message:
$msg = MIME::Lite->new(
    From    => 'me@myhost.com',
    To      => 'you@yourhost.com',
    Cc      => 'some@other.com, some@more.com',
    Subject => 'A message with 2 parts...',
    Type    => 'multipart/mixed'
);

### Add parts (each "attach" has same arguments as "new"):
$msg->attach(Type    => 'TEXT',
            Data     => "Here's the GIF file you wanted"
);
$msg->attach(Type    => 'image/gif',
            Path     => 'aaa000123.gif',
            Filename => 'logo.gif'
);

```

```
$text = $msg->as_string;
```

MIME::Lite also includes a method for sending these things.

```
$msg->send;
```

This defaults to using *sendmail(1)* but can be customized to use SMTP via *Net::SMTP*.

How do I read mail?

While you could use the Mail::Folder module from CPAN (part of the MailFolder package) or the Mail::Internet module from CPAN (also part of the MailTools package), often a module is overkill. Here's a mail sorter.

```
#!/usr/bin/perl
# bysub1 - simple sort by subject
my(@msgs, @sub);
my $msgno = -1;
$/ = '';                                # paragraph reads
while (<>) {
    if (/^From/m) {
        /^Subject:\s*(?:Re:\s*)*(.*)/mi;
        $sub[++$msgno] = lc($1) || '';
    }
    $msgs[$msgno] .= $_;
}
for my $i (sort { $sub[$a] cmp $sub[$b] || $a <=> $b } (0 .. $#msgs)) {
    print $msgs[$i];
}
```

Or more succinctly,

```
#!/usr/bin/perl -n00
# bysub2 - awkish sort-by-subject
BEGIN { $msgno = -1 }
$sub[++$msgno] = (/^Subject:\s*(?:Re:\s*)*(.*)/mi)[0] if /^From/m;
$msgs[$msgno] .= $_;
END { print @msgs[ sort { $sub[$a] cmp $sub[$b] || $a <=> $b } (0 .. $#msg) ] }
```

How do I find out my hostname/domainname/IP address?

The normal way to find your own hostname is to call the `'hostname'` program. While sometimes expedient, this has some problems, such as not knowing whether you've got the canonical name or not. It's one of those tradeoffs of convenience versus portability.

The Sys::Hostname module (part of the standard perl distribution) will give you the hostname after which you can find out the IP address (assuming you have working DNS) with a `gethostbyname()` call.

```
use Socket;
use Sys::Hostname;
my $host = hostname();
my $addr = inet_ntoa(scalar gethostbyname($host || 'localhost'));
```

Probably the simplest way to learn your DNS domain name is to grok it out of `/etc/resolv.conf`, at least under Unix. Of course, this assumes several things about your `resolv.conf` configuration, including that it exists.

(We still need a good DNS domain name-learning method for non-Unix systems.)

How do I fetch a news article or the active newsgroups?

Use the Net::NNTP or News::NNTPClient modules, both available from CPAN. This can make tasks like fetching the newsgroup list as simple as

```
perl -MNews::NNTPClient  
-e 'print News::NNTPClient->new->list("newsgroups")'
```

How do I fetch/put an FTP file?

LWP::Simple (available from CPAN) can fetch but not put. Net::FTP (also available from CPAN) is more complex but can put as well as fetch.

How can I do RPC in Perl?

A DCE::RPC module is being developed (but is not yet available) and will be released as part of the DCE-Perl package (available from CPAN). The rpcgen suite, available from CPAN/authors/id/JAKE/, is an RPC stub generator and includes an RPC::ONC module.

AUTHOR AND COPYRIGHT

Copyright (c) 1997–2002 Tom Christiansen and Nathan Torkington. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

NAME

perlfILTER – Source Filters

DESCRIPTION

This article is about a little-known feature of Perl called *source filters*. Source filters alter the program text of a module before Perl sees it, much as a C preprocessor alters the source text of a C program before the compiler sees it. This article tells you more about what source filters are, how they work, and how to write your own.

The original purpose of source filters was to let you encrypt your program source to prevent casual piracy. This isn't all they can do, as you'll soon learn. But first, the basics.

CONCEPTS

Before the Perl interpreter can execute a Perl script, it must first read it from a file into memory for parsing and compilation. If that script itself includes other scripts with a `use` or `require` statement, then each of those scripts will have to be read from their respective files as well.

Now think of each logical connection between the Perl parser and an individual file as a *source stream*. A source stream is created when the Perl parser opens a file, it continues to exist as the source code is read into memory, and it is destroyed when Perl is finished parsing the file. If the parser encounters a `require` or `use` statement in a source stream, a new and distinct stream is created just for that file.

The diagram below represents a single source stream, with the flow of source from a Perl script file on the left into the Perl parser on the right. This is how Perl normally operates.

```
file -----> parser
```

There are two important points to remember:

1. Although there can be any number of source streams in existence at any given time, only one will be active.
2. Every source stream is associated with only one file.

A source filter is a special kind of Perl module that intercepts and modifies a source stream before it reaches the parser. A source filter changes our diagram like this:

```
file ----> filter ----> parser
```

If that doesn't make much sense, consider the analogy of a command pipeline. Say you have a shell script stored in the compressed file *trial.gz*. The simple pipeline command below runs the script without needing to create a temporary file to hold the uncompressed file.

```
gunzip -c trial.gz | sh
```

In this case, the data flow from the pipeline can be represented as follows:

```
trial.gz ----> gunzip ----> sh
```

With source filters, you can store the text of your script compressed and use a source filter to uncompress it for Perl's parser:

```
compressed          gunzip
Perl program ----> source filter ----> parser
```

USING FILTERS

So how do you use a source filter in a Perl script? Above, I said that a source filter is just a special kind of module. Like all Perl modules, a source filter is invoked with a `use` statement.

Say you want to pass your Perl source through the C preprocessor before execution. You could use the existing `-P` command line option to do this, but as it happens, the source filters distribution comes with a C preprocessor filter module called `Filter::cpp`. Let's use that instead.

Below is an example program, `cpp_test`, which makes use of this filter. Line numbers have been added to allow specific lines to be referenced easily.

```
1: use Filter::cpp ;
2: #define TRUE 1
3: $a = TRUE ;
4: print "a = $a\n" ;
```

When you execute this script, Perl creates a source stream for the file. Before the parser processes any of the lines from the file, the source stream looks like this:

```
cpp test -----> parser
```

Line 1, use `Filter::cpp`, includes and installs the `cpp` filter module. All source filters work this way. The use statement is compiled and executed at compile time, before any more of the file is read, and it attaches the `cpp` filter to the source stream behind the scenes. Now the data flow looks like this:

```
cpp test ----> cpp filter ----> parser
```

As the parser reads the second and subsequent lines from the source stream, it feeds those lines through the `cpp` source filter before processing them. The `cpp` filter simply passes each line through the real C preprocessor. The output from the C preprocessor is then inserted back into the source stream by the filter.

[illegible]

The parser then sees the following code:

```
use Filter::cpp ;
$a = 1 ;
print "a = $a\n" ;
```

Let's consider what happens when the filtered code includes another module with use:

```
1: use Filter::cpp ;
2: #define TRUE 1
3: use Fred ;
4: $a = TRUE ;
5: print "a = $a\n" ;
```

The `cpp` filter does not apply to the text of the `Fred` module, only to the text of the file that used it (`cpp_test`). Although the use statement on line 3 will pass through the `cpp` filter, the module that gets included (`Fred`) will not. The source streams look like this after line 3 has been parsed and before line 4 is parsed:

```
cpp_test ---> cpp filter ---> parser (INACTIVE)
Fred.pm ----> parser
```

As you can see, a new stream has been created for reading the source from `Fred.pm`. This stream will remain active until all of `Fred.pm` has been parsed. The source stream for `cpp_test` will still exist, but is inactive. Once the parser has finished reading `Fred.pm`, the source stream associated with it will be destroyed. The source stream for `cpp_test` then becomes active again and the parser reads line 4 and subsequent lines from `cpp_test`.

You can use more than one source filter on a single file. Similarly, you can reuse the same filter in as many files as you like.

For example, if you have a uuencoded and compressed source file, it is possible to stack a uuencode filter and an uncompression filter like this:

```

use Filter::uudecode ; use Filter::uncompress ;
M'XL("H<US4'V9I;F%L')Q;>7/;1I;_>_I3=&E=%:F*I"T?22Q/
M6]9*<IQCO*XFT"0[PL%%'Y+IG?WN^ZYN-$'J.[.JES,20/?K=_ [>
...

```

Once the first line has been processed, the flow will look like this:

```

file ---> uudecode ---> uncompress ---> parser
      filter           filter

```

Data flows through filters in the same order they appear in the source file. The uudecode filter appeared before the uncompress filter, so the source file will be uudecoded before it's uncompressed.

WRITING A SOURCE FILTER

There are three ways to write your own source filter. You can write it in C, use an external program as a filter, or write the filter in Perl. I won't cover the first two in any great detail, so I'll get them out of the way first. Writing the filter in Perl is most convenient, so I'll devote the most space to it.

WRITING A SOURCE FILTER IN C

The first of the three available techniques is to write the filter completely in C. The external module you create interfaces directly with the source filter hooks provided by Perl.

The advantage of this technique is that you have complete control over the implementation of your filter. The big disadvantage is the increased complexity required to write the filter – not only do you need to understand the source filter hooks, but you also need a reasonable knowledge of Perl guts. One of the few times it is worth going to this trouble is when writing a source scrambler. The `decrypt` filter (which unscrambles the source before Perl parses it) included with the source filter distribution is an example of a C source filter (see [Decryption Filters](#), below).

Decryption Filters

All decryption filters work on the principle of "security through obscurity." Regardless of how well you write a decryption filter and how strong your encryption algorithm, anyone determined enough can retrieve the original source code. The reason is quite simple – once the decryption filter has decrypted the source back to its original form, fragments of it will be stored in the computer's memory as Perl parses it. The source might only be in memory for a short period of time, but anyone possessing a debugger, skill, and lots of patience can eventually reconstruct your program.

That said, there are a number of steps that can be taken to make life difficult for the potential cracker. The most important: Write your decryption filter in C and statically link the decryption module into the Perl binary. For further tips to make life difficult for the potential cracker, see the file *decrypt.pm* in the source filters module.

CREATING A SOURCE FILTER AS A SEPARATE EXECUTABLE

An alternative to writing the filter in C is to create a separate executable in the language of your choice. The separate executable reads from standard input, does whatever processing is necessary, and writes the filtered data to standard output. `Filter::cpp` is an example of a source filter implemented as a separate executable – the executable is the C preprocessor bundled with your C compiler.

The source filter distribution includes two modules that simplify this task: `Filter::exec` and `Filter::sh`. Both allow you to run any external executable. Both use a coprocess to control the flow of data into and out of the external executable. (For details on coprocesses, see Stephens, W.R. "Advanced Programming in the UNIX Environment." Addison-Wesley, ISBN 0-210-56317-7, pages 441-445.) The difference between them is that `Filter::exec` spawns the external command directly, while `Filter::sh` spawns a shell to execute the external command. (Unix uses the Bourne shell; NT uses the `cmd` shell.) Spawning a shell allows you to make use of the shell metacharacters and redirection facilities.

Here is an example script that uses `Filter::sh`:

```

use Filter::sh 'tr XYZ PQR' ;
$a = 1 ;

```

```
print "XYZ a = $a\n" ;
```

The output you'll get when the script is executed:

```
PQR a = 1
```

Writing a source filter as a separate executable works fine, but a small performance penalty is incurred. For example, if you execute the small example above, a separate subprocess will be created to run the Unix `tr` command. Each use of the filter requires its own subprocess. If creating subprocesses is expensive on your system, you might want to consider one of the other options for creating source filters.

WRITING A SOURCE FILTER IN PERL

The easiest and most portable option available for creating your own source filter is to write it completely in Perl. To distinguish this from the previous two techniques, I'll call it a Perl source filter.

To help understand how to write a Perl source filter we need an example to study. Here is a complete source filter that performs rot13 decoding. (Rot13 is a very simple encryption scheme used in Usenet postings to hide the contents of offensive posts. It moves every letter forward thirteen places, so that A becomes N, B becomes O, and Z becomes M.)

```
package Rot13 ;
use Filter::Util::Call ;

sub import {
    my ($type) = @_ ;
    my ($ref) = [] ;
    filter_add(bless $ref) ;
}

sub filter {
    my ($self) = @_ ;
    my ($status) ;

    tr/n-za-mN-ZA-M/a-zA-Z/
    if ($status = filter_read()) > 0 ;
    $status ;
}

1;
```

All Perl source filters are implemented as Perl classes and have the same basic structure as the example above.

First, we include the `Filter::Util::Call` module, which exports a number of functions into your filter's namespace. The filter shown above uses two of these functions, `filter_add()` and `filter_read()`.

Next, we create the filter object and associate it with the source stream by defining the `import` function. If you know Perl well enough, you know that `import` is called automatically every time a module is included with a `use` statement. This makes `import` the ideal place to both create and install a filter object.

In the example filter, the object (`$ref`) is blessed just like any other Perl object. Our example uses an anonymous array, but this isn't a requirement. Because this example doesn't need to store any context information, we could have used a scalar or hash reference just as well. The next section demonstrates context data.

The association between the filter object and the source stream is made with the `filter_add()` function. This takes a filter object as a parameter (`$ref` in this case) and installs it in the source stream.

Finally, there is the code that actually does the filtering. For this type of Perl source filter, all the filtering is done in a method called `filter()`. (It is also possible to write a Perl source filter using a closure. See the `Filter::Util::Call` manual page for more details.) It's called every time the Perl parser needs

another line of source to process. The `filter()` method, in turn, reads lines from the source stream using the `filter_read()` function.

If a line was available from the source stream, `filter_read()` returns a status value greater than zero and appends the line to `$_`. A status value of zero indicates end-of-file, less than zero means an error. The filter function itself is expected to return its status in the same way, and put the filtered line it wants written to the source stream in `$_`. The use of `$_` accounts for the brevity of most Perl source filters.

In order to make use of the `rot13` filter we need some way of encoding the source file in `rot13` format. The script below, `mkrot13`, does just that.

```
die "usage mkrot13 filename\n" unless @ARGV ;
my $in = $ARGV[0] ;
my $out = "$in.tmp" ;
open(IN, "<$in") or die "Cannot open file $in: $!\n";
open(OUT, ">$out") or die "Cannot open file $out: $!\n";

print OUT "use Rot13;\n" ;
while (<IN>) {
    tr/a-zA-Z/n-za-mN-ZA-M/ ;
    print OUT ;
}

close IN;
close OUT;
unlink $in;
rename $out, $in;
```

If we encrypt this with `mkrot13`:

```
print " hello fred \n" ;
```

the result will be this:

```
use Rot13;
cevag "uryyb serq\`a" ;
```

Running it produces this output:

```
hello fred
```

USING CONTEXT: THE DEBUG FILTER

The `rot13` example was a trivial example. Here's another demonstration that shows off a few more features.

Say you wanted to include a lot of debugging code in your Perl script during development, but you didn't want it available in the released product. Source filters offer a solution. In order to keep the example simple, let's say you wanted the debugging output to be controlled by an environment variable, `DEBUG`. Debugging code is enabled if the variable exists, otherwise it is disabled.

Two special marker lines will bracket debugging code, like this:

```
## DEBUG_BEGIN
if ($year > 1999) {
    warn "Debug: millennium bug in year $year\n" ;
}
## DEBUG_END
```

When the `DEBUG` environment variable exists, the filter ensures that Perl parses only the code between the `DEBUG_BEGIN` and `DEBUG_END` markers. That means that when `DEBUG` does exist, the code above should be passed through the filter unchanged. The marker lines can also be passed through as-is, because the Perl parser will see them as comment lines. When `DEBUG` isn't set, we need a way to disable the debug code. A simple way to achieve that is to convert the lines between the two markers into comments:

```
## DEBUG_BEGIN
#if ($year > 1999) {
#    warn "Debug: millennium bug in year $year\n" ;
#}
## DEBUG_END
```

Here is the complete Debug filter:

```
package Debug;

use strict;
use warnings;
use Filter::Util::Call ;

use constant TRUE => 1 ;
use constant FALSE => 0 ;

sub import {
    my ($type) = @_ ;
    my (%context) = (
        Enabled => defined $ENV{DEBUG},
        InTraceBlock => FALSE,
        Filename => (caller)[1],
        LineNo => 0,
        LastBegin => 0,
    ) ;
    filter_add(bless \%context) ;
}

sub Die {
    my ($self) = shift ;
    my ($message) = shift ;
    my ($line_no) = shift || $self->{LastBegin} ;
    die "$message at $self->{Filename} line $line_no.\n"
}

sub filter {
    my ($self) = @_ ;
    my ($status) ;
    $status = filter_read() ;
    ++ $self->{LineNo} ;

    # deal with EOF/error first
    if ($status <= 0) {
        $self->Die("DEBUG_BEGIN has no DEBUG_END")
        if $self->{InTraceBlock} ;
        return $status ;
    }

    if ($self->{InTraceBlock}) {
        if (/^\s*##\s*DEBUG_BEGIN/ ) {
            $self->Die("Nested DEBUG_BEGIN", $self->{LineNo})
        } elsif (/^\s*##\s*DEBUG_END/) {
            $self->{InTraceBlock} = FALSE ;
        }

        # comment out the debug lines when the filter is disabled
        s/^/#/ if ! $self->{Enabled} ;
    } elsif ( /^\s*##\s*DEBUG_BEGIN/ ) {
```

```

        $self->{InTraceBlock} = TRUE ;
        $self->{LastBegin} = $self->{LineNo} ;
    } elsif ( /^\\s*##\\s*DEBUG_END/ ) {
        $self->Die("DEBUG_END has no DEBUG_BEGIN", $self->{LineNo});
    }
    return $status ;
}
1 ;

```

The big difference between this filter and the previous example is the use of context data in the filter object. The filter object is based on a hash reference, and is used to keep various pieces of context information between calls to the filter function. All but two of the hash fields are used for error reporting. The first of those two, Enabled, is used by the filter to determine whether the debugging code should be given to the Perl parser. The second, InTraceBlock, is true when the filter has encountered a DEBUG_BEGIN line, but has not yet encountered the following DEBUG_END line.

If you ignore all the error checking that most of the code does, the essence of the filter is as follows:

```

sub filter {
    my ($self) = @_ ;
    my ($status) ;
    $status = filter_read() ;

    # deal with EOF/error first
    return $status if $status <= 0 ;
    if ($self->{InTraceBlock}) {
        if (/^\\s*##\\s*DEBUG_END/) {
            $self->{InTraceBlock} = FALSE
        }

        # comment out debug lines when the filter is disabled
        s/^/#/ if ! $self->{Enabled} ;
    } elsif ( /^\\s*##\\s*DEBUG_BEGIN/ ) {
        $self->{InTraceBlock} = TRUE ;
    }
    return $status ;
}

```

Be warned: just as the C-preprocessor doesn't know C, the Debug filter doesn't know Perl. It can be fooled quite easily:

```

print <<EOM;
##DEBUG_BEGIN
EOM

```

Such things aside, you can see that a lot can be achieved with a modest amount of code.

CONCLUSION

You now have better understanding of what a source filter is, and you might even have a possible use for them. If you feel like playing with source filters but need a bit of inspiration, here are some extra features you could add to the Debug filter.

First, an easy one. Rather than having debugging code that is all-or-nothing, it would be much more useful to be able to control which specific blocks of debugging code get included. Try extending the syntax for debug blocks to allow each to be identified. The contents of the DEBUG environment variable can then be used to control which blocks get included.

Once you can identify individual blocks, try allowing them to be nested. That isn't difficult either.

Here is an interesting idea that doesn't involve the Debug filter. Currently Perl subroutines have fairly limited support for formal parameter lists. You can specify the number of parameters and their type, but you still have to manually take them out of the `@_` array yourself. Write a source filter that allows you to have a named parameter list. Such a filter would turn this:

```
sub MySub ($first, $second, @rest) { ... }
```

into this:

```
sub MySub($$@) {  
    my ($first) = shift ;  
    my ($second) = shift ;  
    my (@rest) = @_ ;  
    ...  
}
```

Finally, if you feel like a real challenge, have a go at writing a full-blown Perl macro preprocessor as a source filter. Borrow the useful features from the C preprocessor and any other macro processors you know. The tricky bit will be choosing how much knowledge of Perl's syntax you want your filter to have.

REQUIREMENTS

The Source Filters distribution is available on CPAN, in

```
CPAN/modules/by-module/Filter
```

Starting from Perl 5.8 `Filter::Util::Call` (the core part of the Source Filters distribution) is part of the standard Perl distribution. Also included is a friendlier interface called `Filter::Simple`, by Damian Conway.

AUTHOR

Paul Marquess <Paul.Marquess@btinternet.com>

Copyrights

This article originally appeared in The Perl Journal #11, and is copyright 1998 The Perl Journal. It appears courtesy of Jon Orwant and The Perl Journal. This document may be distributed under the same terms as Perl itself.

NAME

perlfork – Perl's fork() emulation

SYNOPSIS

NOTE: As of the 5.8.0 release, fork() emulation has considerably matured. However, there are still a few known bugs and differences from real fork() that might affect you. See the "BUGS" and "CAVEATS AND LIMITATIONS" sections below.

Perl provides a fork() keyword that corresponds to the Unix system call of the same name. On most Unix-like platforms where the fork() system call is available, Perl's fork() simply calls it.

On some platforms such as Windows where the fork() system call is not available, Perl can be built to emulate fork() at the interpreter level. While the emulation is designed to be as compatible as possible with the real fork() at the level of the Perl program, there are certain important differences that stem from the fact that all the pseudo child "processes" created this way live in the same real process as far as the operating system is concerned.

This document provides a general overview of the capabilities and limitations of the fork() emulation. Note that the issues discussed here are not applicable to platforms where a real fork() is available and Perl has been configured to use it.

DESCRIPTION

The fork() emulation is implemented at the level of the Perl interpreter. What this means in general is that running fork() will actually clone the running interpreter and all its state, and run the cloned interpreter in a separate thread, beginning execution in the new thread just after the point where the fork() was called in the parent. We will refer to the thread that implements this child "process" as the pseudo-process.

To the Perl program that called fork(), all this is designed to be transparent. The parent returns from the fork() with a pseudo-process ID that can be subsequently used in any process manipulation functions; the child returns from the fork() with a value of 0 to signify that it is the child pseudo-process.

Behavior of other Perl features in forked pseudo-processes

Most Perl features behave in a natural way within pseudo-processes.

`$$` or `$PROCESS_ID`

This special variable is correctly set to the pseudo-process ID. It can be used to identify pseudo-processes within a particular session. Note that this value is subject to recycling if any pseudo-processes are launched after others have been wait()-ed on.

%ENV Each pseudo-process maintains its own virtual environment. Modifications to %ENV affect the virtual environment, and are only visible within that pseudo-process, and in any processes (or pseudo-processes) launched from it.

chdir() and all other builtins that accept filenames

Each pseudo-process maintains its own virtual idea of the current directory. Modifications to the current directory using chdir() are only visible within that pseudo-process, and in any processes (or pseudo-processes) launched from it. All file and directory accesses from the pseudo-process will correctly map the virtual working directory to the real working directory appropriately.

wait() and **waitpid()**

wait() and waitpid() can be passed a pseudo-process ID returned by fork(). These calls will properly wait for the termination of the pseudo-process and return its status.

kill() kill() can be used to terminate a pseudo-process by passing it the ID returned by fork(). This should not be used except under dire circumstances, because the operating system may not guarantee integrity of the process resources when a running thread is terminated. Note that using kill() on a pseudo-process() may typically cause memory leaks, because the thread

that implements the pseudo-process does not get a chance to clean up its resources.

`exec()` Calling `exec()` within a pseudo-process actually spawns the requested executable in a separate process and waits for it to complete before exiting with the same exit status as that process. This means that the process ID reported within the running executable will be different from what the earlier Perl `fork()` might have returned. Similarly, any process manipulation functions applied to the ID returned by `fork()` will affect the waiting pseudo-process that called `exec()`, not the real process it is waiting for after the `exec()`.

`exit()` `exit()` always exits just the executing pseudo-process, after automatically `wait()`-ing for any outstanding child pseudo-processes. Note that this means that the process as a whole will not exit unless all running pseudo-processes have exited.

Open handles to files, directories and network sockets

All open handles are `dup()`-ed in pseudo-processes, so that closing any handles in one process does not affect the others. See below for some limitations.

Resource limits

In the eyes of the operating system, pseudo-processes created via the `fork()` emulation are simply threads in the same process. This means that any process-level limits imposed by the operating system apply to all pseudo-processes taken together. This includes any limits imposed by the operating system on the number of open file, directory and socket handles, limits on disk space usage, limits on memory size, limits on CPU utilization etc.

Killing the parent process

If the parent process is killed (either using Perl's `kill()` builtin, or using some external means) all the pseudo-processes are killed as well, and the whole process exits.

Lifetime of the parent process and pseudo-processes

During the normal course of events, the parent process and every pseudo-process started by it will wait for their respective pseudo-children to complete before they exit. This means that the parent and every pseudo-child created by it that is also a pseudo-parent will only exit after their pseudo-children have exited.

A way to mark a pseudo-processes as running detached from their parent (so that the parent would not have to `wait()` for them if it doesn't want to) will be provided in future.

CAVEATS AND LIMITATIONS

BEGIN blocks

The `fork()` emulation will not work entirely correctly when called from within a `BEGIN` block. The forked copy will run the contents of the `BEGIN` block, but will not continue parsing the source stream after the `BEGIN` block. For example, consider the following code:

```
BEGIN {
    fork and exit;           # fork child and exit the parent
    print "inner\n";
}
print "outer\n";
```

This will print:

```
inner
```

rather than the expected:

```
inner
outer
```

This limitation arises from fundamental technical difficulties in cloning and restarting the stacks used by the Perl parser in the middle of a parse.

Open filehandles

Any filehandles open at the time of the `fork()` will be `dup()`-ed. Thus, the files can be closed independently in the parent and child, but beware that the `dup()`-ed handles will still share the same seek pointer. Changing the seek position in the parent will change it in the child and vice-versa. One can avoid this by opening files that need distinct seek pointers separately in the child.

Forking pipe `open()` not yet implemented

The `open(FOO, "|-")` and `open(BAR, "-|")` constructs are not yet implemented. This limitation can be easily worked around in new code by creating a pipe explicitly. The following example shows how to write to a forked child:

```
# simulate open(FOO, "|-")
sub pipe_to_fork ($) {
    my $parent = shift;
    pipe my $child, $parent or die;
    my $pid = fork();
    die "fork() failed: $!" unless defined $pid;
    if ($pid) {
        close $child;
    }
    else {
        close $parent;
        open(STDIN, "<&=" . fileno($child)) or die;
    }
    $pid;
}

if (pipe_to_fork('FOO')) {
    # parent
    print FOO "pipe_to_fork\n";
    close FOO;
}
else {
    # child
    while (<STDIN>) { print; }
    close STDIN;
    exit(0);
}
```

And this one reads from the child:

```
# simulate open(FOO, "-|")
sub pipe_from_fork ($) {
    my $parent = shift;
    pipe $parent, my $child or die;
    my $pid = fork();
    die "fork() failed: $!" unless defined $pid;
    if ($pid) {
        close $child;
    }
    else {
        close $parent;
        open(STDOUT, ">&=" . fileno($child)) or die;
    }
    $pid;
}
```

```

    }
    if (pipe_from_fork('BAR')) {
        # parent
        while (<BAR>) { print; }
        close BAR;
    }
    else {
        # child
        print "pipe_from_fork\n";
        close STDOUT;
        exit(0);
    }
}

```

Forking pipe open() constructs will be supported in future.

Global state maintained by XSUBs

External subroutines (XSUBs) that maintain their own global state may not work correctly. Such XSUBs will either need to maintain locks to protect simultaneous access to global data from different pseudo-processes, or maintain all their state on the Perl symbol table, which is copied naturally when fork() is called. A callback mechanism that provides extensions an opportunity to clone their state will be provided in the near future.

Interpreter embedded in larger application

The fork() emulation may not behave as expected when it is executed in an application which embeds a Perl interpreter and calls Perl APIs that can evaluate bits of Perl code. This stems from the fact that the emulation only has knowledge about the Perl interpreter's own data structures and knows nothing about the containing application's state. For example, any state carried on the application's own call stack is out of reach.

Thread-safety of extensions

Since the fork() emulation runs code in multiple threads, extensions calling into non-thread-safe libraries may not work reliably when calling fork(). As Perl's threading support gradually becomes more widely adopted even on platforms with a native fork(), such extensions are expected to be fixed for thread-safety.

BUGS

- Having pseudo-process IDs be negative integers breaks down for the integer -1 because the wait() and waitpid() functions treat this number as being special. The tacit assumption in the current implementation is that the system never allocates a thread ID of 1 for user threads. A better representation for pseudo-process IDs will be implemented in future.
- In certain cases, the OS-level handles created by the pipe(), socket(), and accept() operators are apparently not duplicated accurately in pseudo-processes. This only happens in some situations, but where it does happen, it may result in deadlocks between the read and write ends of pipe handles, or inability to send or receive data across socket handles.
- This document may be incomplete in some respects.

AUTHOR

Support for concurrent interpreters and the fork() emulation was implemented by ActiveState, with funding from Microsoft Corporation.

This document is authored and maintained by Gurusamy Sarathy <gsar@activestate.com>.

SEE ALSO

fork in perlfunc, perlipc

NAME

perlform – Perl formats

DESCRIPTION

Perl has a mechanism to help you generate simple reports and charts. To facilitate this, Perl helps you code up your output page close to how it will look when it's printed. It can keep track of things like how many lines are on a page, what page you're on, when to print page headers, etc. Keywords are borrowed from FORTRAN: `format()` to declare and `write()` to execute; see their entries in [perlfunc](#). Fortunately, the layout is much more legible, more like BASIC's PRINT USING statement. Think of it as a poor man's `nroff(1)`.

Formats, like packages and subroutines, are declared rather than executed, so they may occur at any point in your program. (Usually it's best to keep them all together though.) They have their own namespace apart from all the other "types" in Perl. This means that if you have a function named "Foo", it is not the same thing as having a format named "Foo". However, the default name for the format associated with a given filehandle is the same as the name of the filehandle. Thus, the default format for STDOUT is named "STDOUT", and the default format for filehandle TEMP is named "TEMP". They just look the same. They aren't.

Output record formats are declared as follows:

```
format NAME =  
FORMLIST  
.
```

If name is omitted, format "STDOUT" is defined. FORMLIST consists of a sequence of lines, each of which may be one of three types:

1. A comment, indicated by putting a '#' in the first column.
2. A "picture" line giving the format for one output line.
3. An argument line supplying values to plug into the previous picture line.

Picture lines are printed exactly as they look, except for certain fields that substitute values into the line. Each field in a picture line starts with either "@" (at) or "^" (caret). These lines do not undergo any kind of variable interpolation. The at field (not to be confused with the array marker @) is the normal kind of field; the other kind, caret fields, are used to do rudimentary multi-line text block filling. The length of the field is supplied by padding out the field with multiple "<", ">", or "|" characters to specify, respectively, left justification, right justification, or centering. If the variable would exceed the width specified, it is truncated.

As an alternate form of right justification, you may also use "#" characters (with an optional ".") to specify a numeric field. This way you can line up the decimal points. With a "0" (zero) instead of the first "#", the formatted number will be padded with leading zeroes if necessary. If any value supplied for these fields contains a newline, only the text up to the newline is printed. Finally, the special field "@*" can be used for printing multi-line, nontruncated values; it should appear by itself on a line.

The values are specified on the following line in the same order as the picture fields. The expressions providing the values should be separated by commas. The expressions are all evaluated in a list context before the line is processed, so a single list expression could produce multiple list elements. The expressions may be spread out to more than one line if enclosed in braces. If so, the opening brace must be the first token on the first line. If an expression evaluates to a number with a decimal part, and if the corresponding picture specifies that the decimal part should appear in the output (that is, any picture except multiple "#" characters **without** an embedded "."), the character used for the decimal point is **always** determined by the current LC_NUMERIC locale. This means that, if, for example, the run-time environment happens to specify a German locale, "," will be used instead of the default ".". See [perllocale](#) and ["WARNINGS"](#) for more information.

Picture fields that begin with ^ rather than @ are treated specially. With a # field, the field is blanked out if the value is undefined. For other field types, the caret enables a kind of fill mode. Instead of an arbitrary expression, the value supplied must be a scalar variable name that contains a text string. Perl puts as much text as it can into the field, and then chops off the front of the string so that the next time the variable is referenced, more of the text can be printed. (Yes, this means that the variable itself is altered during execution of the `write()` call, and is not returned.) Normally you would use a sequence of fields in a vertical stack to print out a block of text. You might wish to end the final field with the text "...", which will appear in the output if the text was too long to appear in its entirety. You can change which characters are legal to break on by changing the variable `$:` (that's `$FORMAT_LINE_BREAK_CHARACTERS` if you're using the English module) to a list of the desired characters.

Using caret fields can produce variable length records. If the text to be formatted is short, you can suppress blank lines by putting a "~" (tilde) character anywhere in the line. The tilde will be translated to a space upon output. If you put a second tilde contiguous to the first, the line will be repeated until all the fields on the line are exhausted. (If you use a field of the at variety, the expression you supply had better not give the same value every time forever!)

Top-of-form processing is by default handled by a format with the same name as the current filehandle with "_TOP" concatenated to it. It's triggered at the top of each page. See [write](#).

Examples:

[illegible]

```

~                                     ^<<<<<<<<<<<<<<<<<<<<<<<<<<<
~                                     $description
~                                     ^<<<<<<<<<<<<<<<<<<<<<<<<<<<...
~                                     $description
.
```

It is possible to intermix `print()`s with `write()`s on the same output channel, but you'll have to handle `$-($FORMAT LINES LEFT)` yourself.

Format Variables

The current format name is stored in the variable `$~ ($FORMAT_NAME)`, and the current top of form format name is in `$^ ($FORMAT_TOP_NAME)`. The current output page number is stored in `$% ($FORMAT_PAGE_NUMBER)`, and the number of lines on the page is in `$= ($FORMAT_LINES_PER_PAGE)`. Whether to autoflush output on this handle is stored in `$| ($OUTPUT_AUTOFLUSH)`. The string output before each top of page (except the first) is stored in `$^L ($FORMAT_FORMFEED)`. These variables are set on a per-filehandle basis, so you'll need to `select()` into a different one to affect them:

```
select((select(OUTF),
           $~ = "My_Other_Format",
           $^ = "My_Top_Format"
        ) [0]);
```

Pretty ugly, eh? It's a common idiom though, so don't be too surprised when you see it. You can at least use a temporary variable to hold the previous filehandle: (this is a much better approach in general, because not only does legibility improve, you now have intermediary stage in the expression to single-step the debugger through):

```
$ofh = select(OUTF);
$~ = "My_Other_Format";
$^ = "My_Top_Format";
select($ofh);
```

If you use the English module, you can even read the variable names:

```
use English '-no_match_vars';
$ofh = select(OUTF);
$FORMAT_NAME      = "My_Other_Format";
$FORMAT_TOP_NAME  = "My_Top_Format";
select($ofh);
```

But you still have those funny `select()`s. So just use the `FileHandle` module. Now, you can access these special variables using lowercase method names instead:

```
use FileHandle;
format_name      OUTF "My_Other_Format";
format top name OUTF "My Top Format";
```

Much better!

NOTES

Because the values line may contain arbitrary expressions (for `at` fields, not `caret` fields), you can farm out more sophisticated processing to other functions, like `sprintf()` or one of your own. For example:

[illegible]

To get a real at or caret into the field, do this:

Accessing Formatting Internals

For low-level access to the formatting mechanism, you may use `formline()` and access `$$A` (the `$ACCUMULATOR` variable) directly.

For example:

```
$str = formline <<'END', 1,2,3;
@<<< @||| @>>>
END

print "Wow, I just stored '$A' in the accumulator!\n";
```

Or to make an `swrite()` subroutine, which is to write() what `sprintf()` is to `printf()`, do this:

```
use Carp;
sub swrite {
    croak "usage: swrite PICTURE ARGS" unless @_ ;
    my $format = shift;
    $$A = "";
    formline($format,@_);
    return $$A;
}

$string = swrite(<<'END', 1, 2, 3);
Check me out
@<<< @||| @>>>
END
print $string;
```

WARNINGS

The lone dot that ends a format can also prematurely end a mail message passing through a misconfigured Internet mailer (and based on experience, such misconfiguration is the rule, not the exception). So when sending format code through mail, you should indent it so that the format-ending dot is not on the left margin; this will prevent SMTP cutoff.

Lexical variables (declared with "my") are not visible within a format unless the format is declared within the scope of the lexical variable. (They weren't visible at all before version 5.001.)

Formats are the only part of Perl that unconditionally use information from a program's locale; if a program's environment specifies an `LC_NUMERIC` locale, it is always used to specify the decimal point character in formatted output. Perl ignores all other aspects of locale handling unless the `use locale` pragma is in effect. Formatted output cannot be controlled by `use locale` because the pragma is tied to the block structure of the program, and, for historical reasons, formats exist outside that block structure. See [perllocale](#) for further discussion of locale handling.

Inside of an expression, the whitespace characters `\n`, `\t` and `\f` are considered to be equivalent to a single space. Thus, you could think of this filter being applied to each value in the format:

```
$value =~ tr/\n\t\f/ /;
```

The remaining whitespace character, `\r`, forces the printing of a new line if allowed by the picture line.

NAME

perlfunc – Perl builtin functions

DESCRIPTION

The functions in this section can serve as terms in an expression. They fall into two major categories: list operators and named unary operators. These differ in their precedence relationship with a following comma. (See the precedence table in [perlop](#).) List operators take more than one argument, while unary operators can never take more than one argument. Thus, a comma terminates the argument of a unary operator, but merely separates the arguments of a list operator. A unary operator generally provides a scalar context to its argument, while a list operator may provide either scalar or list contexts for its arguments. If it does both, the scalar arguments will be first, and the list argument will follow. (Note that there can ever be only one such list argument.) For instance, `splice()` has three scalar arguments followed by a list, whereas `gethostbyname()` has four scalar arguments.

In the syntax descriptions that follow, list operators that expect a list (and provide list context for the elements of the list) are shown with LIST as an argument. Such a list may consist of any combination of scalar arguments or list values; the list values will be included in the list as if each individual element were interpolated at that point in the list, forming a longer single-dimensional list value. Elements of the LIST should be separated by commas.

Any function in the list below may be used either with or without parentheses around its arguments. (The syntax descriptions omit the parentheses.) If you use the parentheses, the simple (but occasionally surprising) rule is this: It *looks* like a function, therefore it *is* a function, and precedence doesn't matter. Otherwise it's a list operator or unary operator, and precedence does matter. And whitespace between the function and left parenthesis doesn't count—so you need to be careful sometimes:

```
print 1+2+4;      # Prints 7.
print(1+2) + 4;   # Prints 3.
print (1+2)+4;    # Also prints 3!
print +(1+2)+4;   # Prints 7.
print ((1+2)+4);  # Prints 7.
```

If you run Perl with the `-w` switch it can warn you about this. For example, the third line above produces:

```
print (...) interpreted as function at - line 1.
Useless use of integer addition in void context at - line 1.
```

A few functions take no arguments at all, and therefore work as neither unary nor list operators. These include such functions as `time` and `endpwent`. For example, `time+86_400` always means `time() + 86_400`.

For functions that can be used in either a scalar or list context, nonabortive failure is generally indicated in a scalar context by returning the undefined value, and in a list context by returning the null list.

Remember the following important rule: There is **no rule** that relates the behavior of an expression in list context to its behavior in scalar context, or vice versa. It might do two totally different things. Each operator and function decides which sort of value it would be most appropriate to return in scalar context. Some operators return the length of the list that would have been returned in list context. Some operators return the first value in the list. Some operators return the last value in the list. Some operators return a count of successful operations. In general, they do what you want, unless you want consistency.

A named array in scalar context is quite different from what would at first glance appear to be a list in scalar context. You can't get a list like `(1, 2, 3)` into being in scalar context, because the compiler knows the context at compile time. It would generate the scalar comma operator there, not the list construction version of the comma. That means it was never a list to start with.

In general, functions in Perl that serve as wrappers for system calls of the same name (like `chown(2)`, `fork(2)`, `closedir(2)`, etc.) all return true when they succeed and `undef` otherwise, as is usually mentioned in the descriptions below. This is different from the C interfaces, which return `-1` on failure. Exceptions to this

rule are `wait`, `waitpid`, and `syscall`. System calls also set the special `$!` variable on failure. Other functions do not, except accidentally.

Perl Functions by Category

Here are Perl's functions (including things that look like functions, like some keywords and named operators) arranged by category. Some functions appear in more than one place.

Functions for SCALARs or strings

`chomp`, `chop`, `chr`, `crypt`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`,
`q/STRING/`, `qq/STRING/`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`,
`y///`

Regular expressions and pattern matching

`m//`, `pos`, `quotemeta`, `s///`, `split`, `study`, `qr//`

Numeric functions

`abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

Functions for real @ARRAYs

`pop`, `push`, `shift`, `splice`, `unshift`

Functions for list data

`grep`, `join`, `map`, `qw/STRING/`, `reverse`, `sort`, `unpack`

Functions for real %HASHes

`delete`, `each`, `exists`, `keys`, `values`

Input and output functions

`binmode`, `close`, `closedir`, `dbmclose`, `dbmopen`, `die`, `eof`, `fileno`, `flock`, `format`,
`getc`, `print`, `printf`, `read`, `readdir`, `rewinddir`, `seek`, `seekdir`, `select`, `syscall`,
`sysread`, `sysseek`, `syswrite`, `tell`, `telldir`, `truncate`, `warn`, `write`

Functions for fixed length data or records

`pack`, `read`, `syscall`, `sysread`, `syswrite`, `unpack`, `vec`

Functions for filehandles, files, or directories

`-X`, `chdir`, `chmod`, `chown`, `chroot`, `fcntl`, `glob`, `ioctl`, `link`, `lstat`, `mkdir`, `open`,
`opendir`, `readlink`, `rename`, `rmdir`, `stat`, `symlink`, `sysopen`, `umask`, `unlink`, `utime`

Keywords related to the control flow of your perl program

`caller`, `continue`, `die`, `do`, `dump`, `eval`, `exit`, `goto`, `last`, `next`, `redo`, `return`, `sub`,
`wantarray`

Keywords related to scoping

`caller`, `import`, `local`, `my`, `our`, `package`, `use`

Miscellaneous functions

`defined`, `dump`, `eval`, `formline`, `local`, `my`, `our`, `reset`, `scalar`, `undef`, `wantarray`

Functions for processes and process groups

`alarm`, `exec`, `fork`, `getpgrp`, `getppid`, `getpriority`, `kill`, `pipe`, `qx/STRING/`,
`setpgrp`, `setpriority`, `sleep`, `system`, `times`, `wait`, `waitpid`

Keywords related to perl modules

`do`, `import`, `no`, `package`, `require`, `use`

Keywords related to classes and object-orientedness

`bless`, `dbmclose`, `dbmopen`, `package`, `ref`, `tie`, `tied`, `untie`, `use`

Low-level socket functions

accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair

System V interprocess communication functions

msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

Fetching user and group info

endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent

Fetching network info

endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent

Time-related functions

gmtime, localtime, time, times

Functions new in perl5

abs, bless, chomp, chr, exists, formline, glob, import, lc, lcfirst, map, my, no, our, prototype, qx, qw, readline, readpipe, ref, sub*, sysopen, tie, tied, uc, ucfirst, untie, use

* – sub was a keyword in perl4, but in perl5 it is an operator, which can be used in expressions.

Functions obsoleted in perl5

dbmclose, dbmopen

Portability

Perl was born in Unix and can therefore access all common Unix system calls. In non-Unix environments, the functionality of some Unix system calls may not be available, or details of the available functionality may differ slightly. The Perl functions affected by this are:

-X, binmode, chmod, chown, chroot, crypt, dbmclose, dbmopen, dump, endgrent, endhostent, endnetent, endprotoent, endpwent, endservent, exec, fcntl, flock, fork, getgrent, getgrgid, gethostent, getlogin, getnetbyaddr, getnetbyname, getnetent, getppid, getprgp, getpriority, getprotobynumber, getprotoent, getpwent, getpwnam, getpwuid, getservbyport, getservent, getsockopt, glob, ioctl, kill, link, lstat, msgctl, msgget, msgrcv, msgsnd, open, pipe, readlink, rename, select, semctl, semget, semop, setgrent, sethostent, setnetent, setpgrp, setpriority, setprotoent, setpwent, setservent, setsockopt, shmctl, shmget, shmread, shmwrite, socket, socketpair, stat, symlink, syscall, sysopen, system, times, truncate, umask, unlink, utime, wait, waitpid

For more information about the portability of these functions, see [perlport](#) and other available platform-specific documentation.

Alphabetical Listing of Perl Functions

-X FILEHANDLE

-X EXPR

-X A file test, where X is one of the letters listed below. This unary operator takes one argument, either a filename or a filehandle, and tests the associated file to see if something is true about it. If the argument is omitted, tests \$_, except for -t, which tests STDIN. Unless otherwise documented, it returns 1 for true and '' for false, or the undefined value if the file doesn't exist.

Despite the funny names, precedence is the same as any other named unary operator, and the argument may be parenthesized like any other unary operator. The operator may be any of: X<-rX<-wX<-xX<-oX<-RX<-WX<-XX<-OX<-eX<-zX<-sX<-fX<-dX<-lX<-pX<-SX<-bX<-cX<-tX<-uX<-gX<-kX<-TX<-BX<-MX<-AX<-C

- r File is readable by effective uid/gid.
- w File is writable by effective uid/gid.
- x File is executable by effective uid/gid.
- o File is owned by effective uid.
- R File is readable by real uid/gid.
- W File is writable by real uid/gid.
- X File is executable by real uid/gid.
- O File is owned by real uid.
- e File exists.
- z File has zero size (is empty).
- s File has nonzero size (returns size in bytes).
- f File is a plain file.
- d File is a directory.
- l File is a symbolic link.
- p File is a named pipe (FIFO), or Filehandle is a pipe.
- S File is a socket.
- b File is a block special file.
- c File is a character special file.
- t Filehandle is opened to a tty.
- u File has setuid bit set.
- g File has setgid bit set.
- k File has sticky bit set.
- T File is an ASCII text file (heuristic guess).
- B File is a "binary" file (opposite of -T).
- M Script start time minus file modification time, in days.
- A Same for access time.
- C Same for inode change time (Unix, may differ for other platforms)

Example:

```
while (<>) {
    chomp;
    next unless -f $_;      # ignore specials
    #...
}
```

The interpretation of the file permission operators `-r`, `-R`, `-w`, `-W`, `-x`, and `-X` is by default based solely on the mode of the file and the uids and gids of the user. There may be other reasons you can't actually read, write, or execute the file. Such reasons may be for example network filesystem access controls, ACLs (access control lists), read-only filesystems, and unrecognized executable formats.

Also note that, for the superuser on the local filesystems, the `-r`, `-R`, `-w`, and `-W` tests always return 1, and `-x` and `-X` return 1 if any execute bit is set in the mode. Scripts run by the superuser may thus need to do a `stat()` to determine the actual mode of the file, or temporarily set their effective uid to something else.

If you are using ACLs, there is a pragma called `filetest` that may produce more accurate results than the bare `stat()` mode bits. When under the use `filetest 'access'` the

above-mentioned filetests will test whether the permission can (not) be granted using the `access()` family of system calls. Also note that the `-x` and `-X` may under this pragma return true even if there are no execute permission bits set (nor any extra execute permission ACLs). This strangeness is due to the underlying system calls' definitions. Read the documentation for the `filetest` pragma for more information.

Note that `-s/a/b/` does not do a negated substitution. Saying `-exp($foo)` still works as expected, however—only single letters following a minus are interpreted as file tests.

The `-T` and `-B` switches work as follows. The first block or so of the file is examined for odd characters such as strange control codes or characters with the high bit set. If too many strange characters (30%) are found, it's a `-B` file, otherwise it's a `-T` file. Also, any file containing null in the first block is considered a binary file. If `-T` or `-B` is used on a filehandle, the current IO buffer is examined rather than the first block. Both `-T` and `-B` return true on a null file, or a file at EOF when testing a filehandle. Because you have to read a file to do the `-T` test, on most occasions you want to use a `-f` against the file first, as in `next unless -f $file && -T $file`.

If any of the file tests (or either the `stat` or `lstat` operators) are given the special filehandle consisting of a solitary underline, then the `stat` structure of the previous file test (or `stat` operator) is used, saving a system call. (This doesn't work with `-t`, and you need to remember that `lstat()` and `-l` will leave values in the `stat` structure for the symbolic link, not the real file.) (Also, if the `stat` buffer was filled by a `lstat` call, `-T` and `-B` will reset it with the results of `stat _`). Example:

```
print "Can do.\n" if -r $a || -w _ || -x _;
stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;
print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;
```

abs VALUE

abs Returns the absolute value of its argument. If `VALUE` is omitted, uses `$_`.

accept NEWSOCKET,GENERICSOCKET

Accepts an incoming socket connect, just as the `accept(2)` system call does. Returns the packed address if it succeeded, false otherwise. See the example in [Sockets: Client/Server Communication in perlipc](#).

On systems that support a `close-on-exec` flag on files, the flag will be set for the newly opened file descriptor, as determined by the value of `$_^F`. See [\\$^F](#).

alarm SECONDS

alarm Arranges to have a `SIGALRM` delivered to this process after the specified number of wallclock seconds have elapsed. If `SECONDS` is not specified, the value stored in `$_` is used. (On some machines, unfortunately, the elapsed time may be up to one second less or more than you specified because of how seconds are counted, and process scheduling may delay the delivery of the signal even further.)

Only one timer may be counting at once. Each call disables the previous timer, and an argument of `0` may be supplied to cancel the previous timer without starting a new one. The returned value is the amount of time remaining on the previous timer.

For delays of finer granularity than one second, you may use Perl's four-argument version of `select()` leaving the first three arguments undefined, or you might be able to use the `syscall` interface to access `setitimer(2)` if your system supports it. The `Time::HiRes` module (from CPAN, and starting from Perl 5.8 part of the standard distribution) may also prove useful.

It is usually a mistake to intermix `alarm` and `sleep` calls. (`sleep` may be internally implemented in your system with `alarm`)

If you want to use `alarm` to time out a system call you need to use an `eval/die` pair. You can't rely on the alarm causing the system call to fail with `$!` set to `EINTR` because Perl sets up signal handlers to restart system calls on some systems. Using `eval/die` always works, modulo the caveats given in [Signals in perlipc](#).

```
eval {
    local $SIG{ALRM} = sub { die "alarm\n" }; # NB: \n required
    alarm $timeout;
    $nread = sysread SOCKET, $buffer, $size;
    alarm 0;
};
if ($?) {
    die unless $? eq "alarm\n"; # propagate unexpected errors
    # timed out
}
else {
    # didn't
}
```

atan2 Y,X

Returns the arctangent of Y/X in the range $-\pi$ to π .

For the tangent operation, you may use the `Math::Trig::tan` function, or use the familiar relation:

```
sub tan { sin($_[0]) / cos($_[0]) }
```

bind SOCKET,NAME

Binds a network address to a socket, just as the `bind` system call does. Returns true if it succeeded, false otherwise. `NAME` should be a packed address of the appropriate type for the socket. See the examples in [Sockets: Client/Server Communication in perlipc](#).

binmode FILEHANDLE, LAYER

binmode FILEHANDLE

Arranges for `FILEHANDLE` to be read or written in "binary" or "text" mode on systems where the run-time libraries distinguish between binary and text files. If `FILEHANDLE` is an expression, the value is taken as the name of the filehandle. Returns true on success, undef on failure.

If `LAYER` is omitted or specified as `:raw` the filehandle is made suitable for passing binary data. This includes turning off possible CRLF translation and marking it as bytes (as opposed to Unicode characters). Note that as despite what may be implied in *"Programming Perl"* (the Camel) or elsewhere `:raw` is *not* the simply inverse of `:crlf` — other layers which would affect binary nature of the stream are *also* disabled. See [PerlIO](#), [perlrun](#) and the discussion about the `PERLIO` environment variable.

The `LAYER` parameter of the `binmode()` function is described as "DISCIPLINE" in "Programming Perl, 3rd Edition". However, since the publishing of this book, by many known as "Camel III", the consensus of the naming of this functionality has moved from "discipline" to "layer". All documentation of this version of Perl therefore refers to "layers" rather than to "disciplines". Now back to the regularly scheduled documentation...

On some systems (in general, DOS and Windows-based systems) `binmode()` is necessary when you're not working with a text file. For the sake of portability it is a good idea to always use it when appropriate, and to never use it when it isn't appropriate.

In other words: regardless of platform, use `binmode()` on binary files (like for example images).

If `LAYER` is present it is a single string, but may contain multiple directives. The directives alter the behaviour of the file handle. When `LAYER` is present using `binmode` on text file makes sense.

To mark `FILEHANDLE` as UTF-8, use `:utf8`.

The `:bytes`, `:crlf`, and `:utf8`, and any other directives of the form `:... ,` are called I/O *layers*. The `open` pragma can be used to establish default I/O layers. See [open](#).

In general, `binmode()` should be called after `open()` but before any I/O is done on the filehandle. Calling `binmode()` will normally flush any pending buffered output data (and perhaps pending input data) on the handle. An exception to this is the `:encoding` layer that changes the default character encoding of the handle, see [open](#). The `:encoding` layer sometimes needs to be called in mid-stream, and it doesn't flush the stream.

The operating system, device drivers, C libraries, and Perl run-time system all work together to let the programmer treat a single character (`\n`) as the line terminator, irrespective of the external representation. On many operating systems, the native text file representation matches the internal representation, but on some platforms the external representation of `\n` is made up of more than one character.

Mac OS, all variants of Unix, and Stream_LF files on VMS use a single character to end each line in the external representation of text (even though that single character is CARRIAGE RETURN on Mac OS and LINE FEED on Unix and most VMS files). In other systems like OS/2, DOS and the various flavors of MS-Windows your program sees a `\n` as a simple `\cJ`, but what's stored in text files are the two characters `\cM\cJ`. That means that, if you don't use `binmode()` on these systems, `\cM\cJ` sequences on disk will be converted to `\n` on input, and any `\n` in your program will be converted back to `\cM\cJ` on output. This is what you want for text files, but it can be disastrous for binary files.

Another consequence of using `binmode()` (on some systems) is that special end-of-file markers will be seen as part of the data stream. For systems from the Microsoft family this means that if your binary data contains `\cZ`, the I/O subsystem will regard it as the end of the file, unless you use `binmode()`.

`binmode()` is not only important for `readline()` and `print()` operations, but also when using `read()`, `seek()`, `sysread()`, `syswrite()` and `tell()` (see [perlport](#) for more details). See the `$/` and `$\` variables in [perlvar](#) for how to manually set your input and output line-termination sequences.

bless REF,CLASSNAME
bless REF

This function tells the thingy referenced by `REF` that it is now an object in the `CLASSNAME` package. If `CLASSNAME` is omitted, the current package is used. Because a `bless` is often the last thing in a constructor, it returns the reference for convenience. Always use the two-argument version if the function doing the blessing might be inherited by a derived class. See [perltoot](#) and [perlobj](#) for more about the blessing (and blessings) of objects.

Consider always blessing objects in `CLASSNAME`s that are mixed case. Namespaces with all lowercase names are considered reserved for Perl pragmata. Builtin types have all uppercase names, so to prevent confusion, you may wish to avoid such package names as well. Make sure that `CLASSNAME` is a true value.

See *Perl Modules in perlmod*.

caller EXPR

caller Returns the context of the current subroutine call. In scalar context, returns the caller's package name if there is a caller, that is, if we're in a subroutine or `eval` or `require`, and the undefined value otherwise. In list context, returns

```
($package, $filename, $line) = caller;
```

With EXPR, it returns some extra information that the debugger uses to print a stack trace. The value of EXPR indicates how many call frames to go back before the current one.

```
($package, $filename, $line, $subroutine, $hasargs,
 $wantarray, $evaltext, $is_require, $hints, $bitmask) = caller($i);
```

Here `$subroutine` may be `(eval)` if the frame is not a subroutine call, but an `eval`. In such a case additional elements `$evaltext` and `$is_require` are set: `$is_require` is true if the frame is created by a `require` or `use` statement, `$evaltext` contains the text of the `eval EXPR` statement. In particular, for an `eval BLOCK` statement, `$filename` is `(eval)`, but `$evaltext` is undefined. (Note also that each `use` statement creates a `require` frame inside an `eval EXPR` frame.) `$subroutine` may also be `(unknown)` if this particular subroutine happens to have been deleted from the symbol table. `$hasargs` is true if a new instance of `@_` was set up for the frame. `$hints` and `$bitmask` contain pragmatic hints that the caller was compiled with. The `$hints` and `$bitmask` values are subject to change between versions of Perl, and are not meant for external use.

Furthermore, when called from within the DB package, `caller` returns more detailed information: it sets the list variable `@DB: :args` to be the arguments with which the subroutine was invoked.

Be aware that the optimizer might have optimized call frames away before `caller` had a chance to get the information. That means that `caller(N)` might not return information about the call frame you expect it do, for `< N - 1`. In particular, `@DB: :args` might have information from the previous time `caller` was called.

chdir EXPR

Changes the working directory to EXPR, if possible. If EXPR is omitted, changes to the directory specified by `$ENV{HOME}`, if set; if not, changes to the directory specified by `$ENV{LOGDIR}`. (Under VMS, the variable `$ENV{SYS$LOGIN}` is also checked, and used if it is set.) If neither is set, `chdir` does nothing. It returns true upon success, false otherwise. See the example under `die`.

chmod LIST

Changes the permissions of a list of files. The first element of the list must be the numerical mode, which should probably be an octal number, and which definitely should *not* a string of octal digits: 0644 is okay, '0644' is not. Returns the number of files successfully changed. See also */oct*, if all you have is a string.

```
$cnt = chmod 0755, 'foo', 'bar';
chmod 0755, @executables;
$mode = '0644'; chmod $mode, 'foo';      # !!! sets mode to
                                           # --w---r-T
$mode = '0644'; chmod oct($mode), 'foo'; # this is better
$mode = 0644;   chmod $mode, 'foo';      # this is best
```

You can also import the symbolic `S_I*` constants from the `Fcntl` module:

```
use Fcntl ':mode';

chmod S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH, @executables;
# This is identical to the chmod 0755 of the above example.
```


chomp VARIABLE

chomp(LIST)

chomp This safer version of */chop* removes any trailing string that corresponds to the current value of `$/` (also known as `$INPUT_RECORD_SEPARATOR` in the *English* module). It returns the total number of characters removed from all its arguments. It's often used to remove the newline from the end of an input record when you're worried that the final record may be missing its newline. When in paragraph mode (`$/ = ""`), it removes all trailing newlines from the string. When in slurp mode (`$/ = undef`) or fixed-length record mode (`$/` is a reference to an integer or the like, see *perlvar*) `chomp()` won't remove anything. If *VARIABLE* is omitted, it chops `$_`. Example:

```
while (<>) {
    chomp; # avoid \n on last field
    @array = split(/:/);
    # ...
}
```

If *VARIABLE* is a hash, it chops the hash's values, but not its keys.

You can actually `chomp` anything that's an lvalue, including an assignment:

```
chomp($cwd = `pwd`);
chomp($answer = <STDIN>);
```

If you `chomp` a list, each element is chopped, and the total number of characters removed is returned.

Note that parentheses are necessary when you're chopping anything that is not a simple variable. This is because `chomp $cwd = `pwd``; is interpreted as `(chomp $cwd) = `pwd``; , rather than as `chomp($cwd = `pwd`)` which you might expect. Similarly, `chomp $a, $b` is interpreted as `chomp($a), $b` rather than as `chomp($a, $b)` .

chop VARIABLE

chop(LIST)

chop Chops off the last character of a string and returns the character chopped. It is much more efficient than `s/.$//s` because it neither scans nor copies the string. If *VARIABLE* is omitted, chops `$_`. If *VARIABLE* is a hash, it chops the hash's values, but not its keys.

You can actually `chop` anything that's an lvalue, including an assignment.

If you `chop` a list, each element is chopped. Only the value of the last `chop` is returned.

Note that `chop` returns the last character. To return all but the last character, use `substr($string, 0, -1)`.

See also */chomp*.

chown LIST

Changes the owner (and group) of a list of files. The first two elements of the list must be the *numeric* uid and gid, in that order. A value of `-1` in either position is interpreted by most systems to leave that value unchanged. Returns the number of files successfully changed.

```
$cnt = chown $uid, $gid, 'foo', 'bar';
chown $uid, $gid, @filenames;
```

Here's an example that looks up nonnumeric uids in the `passwd` file:

```
print "User: ";
chomp($user = <STDIN>);
print "Files: ";
chomp($pattern = <STDIN>);
```

```

($login,$pass,$uid,$gid) = getpwnam($user)
    or die "$user not in passwd file";

@ary = glob($pattern);      # expand filenames
chown $uid, $gid, @ary;

```

On most systems, you are not allowed to change the ownership of the file unless you're the superuser, although you should be able to change the group to any of your secondary groups. On insecure systems, these restrictions may be relaxed, but this is not a portable assumption. On POSIX systems, you can detect this condition this way:

```

use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
$can_chown_giveaway = not sysconf(_PC_CHOWN_RESTRICTED);

```

chr NUMBER

chr Returns the character represented by that NUMBER in the character set. For example, `chr(65)` is "A" in either ASCII or Unicode, and `chr(0x263a)` is a Unicode smiley face. Note that characters from 127 to 255 (inclusive) are by default not encoded in Unicode for backward compatibility reasons (but see [encoding](#)).

For the reverse, use [/ord](#). See [perlunicode](#) and [encoding](#) for more about Unicode.

If NUMBER is omitted, uses `$_`.

chroot FILENAME

chroot This function works like the system call by the same name: it makes the named directory the new root directory for all further pathnames that begin with a / by your process and all its children. (It doesn't change your current working directory, which is unaffected.) For security reasons, this call is restricted to the superuser. If FILENAME is omitted, does a `chroot` to `$_`.

close FILEHANDLE

close Closes the file or pipe associated with the file handle, returning true only if IO buffers are successfully flushed and closes the system file descriptor. Closes the currently selected filehandle if the argument is omitted.

You don't have to close FILEHANDLE if you are immediately going to do another open on it, because open will close it for you. (See [open](#).) However, an explicit close on an input file resets the line counter (`$.`), while the implicit close done by open does not.

If the file handle came from a piped open close will additionally return false if one of the other system calls involved fails or if the program exits with non-zero status. (If the only problem was that the program exited non-zero `$?` will be set to `.`) Closing a pipe also waits for the process executing on the pipe to complete, in case you want to look at the output of the pipe afterwards, and implicitly puts the exit status value of that command into `$?` .

Prematurely closing the read end of a pipe (i.e. before the process writing to it at the other end has closed it) will result in a SIGPIPE being delivered to the writer. If the other end can't handle that, be sure to read all the data before closing the pipe.

Example:

```

open(OUTPUT, '|sort >foo') # pipe to sort
    or die "Can't start sort: $!";
#...                        # print stuff to output
close OUTPUT                # wait for sort to finish
    or warn $! ? "Error closing sort pipe: $!"
    : "Exit status $? from sort";
open(INPUT, 'foo')          # get sort's results
    or die "Can't open 'foo' for input: $!";

```

FILEHANDLE may be an expression whose value can be used as an indirect filehandle, usually

the real filehandle name.

closedir DIRHANDLE

Closes a directory opened by `opendir` and returns the success of that system call.

DIRHANDLE may be an expression whose value can be used as an indirect dirhandle, usually the real dirhandle name.

connect SOCKET,NAME

Attempts to connect to a remote socket, just as the `connect` system call does. Returns true if it succeeded, false otherwise. NAME should be a packed address of the appropriate type for the socket. See the examples in *Sockets: Client/Server Communication in perlipc*.

continue BLOCK

Actually a flow control statement rather than a function. If there is a `continue BLOCK` attached to a BLOCK (typically in a `while` or `foreach`), it is always executed just before the conditional is about to be evaluated again, just like the third part of a `for` loop in C. Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement (which is similar to the C `continue` statement).

`last`, `next`, or `redo` may appear within a `continue` block. `last` and `redo` will behave as if they had been executed within the main block. So will `next`, but since it will execute a `continue` block, it may be more entertaining.

```
while (EXPR) {
    ### redo always comes here
    do_something;
} continue {
    ### next always comes here
    do_something_else;
    # then back the top to re-check EXPR
}
### last always comes here
```

Omitting the `continue` section is semantically equivalent to using an empty one, logically enough. In that case, `next` goes directly back to check the condition at the top of the loop.

cos EXPR

cos Returns the cosine of EXPR (expressed in radians). If EXPR is omitted, takes cosine of `$_`.

For the inverse cosine operation, you may use the `Math::Trig::acos()` function, or use this relation:

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }
```

crypt PLAINTEXT,SALT

Encrypts a string exactly like the `crypt(3)` function in the C library (assuming that you actually have a version there that has not been extirpated as a potential munition). This can prove useful for checking the password file for lousy passwords, amongst other things. Only the guys wearing white hats should do this.

Note that `crypt` is intended to be a one-way function, much like breaking eggs to make an omelette. There is no (known) corresponding decrypt function (in other words, the `crypt()` is a one-way hash function). As a result, this function isn't all that useful for cryptography. (For that, see your nearby CPAN mirror.)

When verifying an existing encrypted string you should use the encrypted text as the salt (like `crypt($plain, $crypteq) eq $crypteq`). This allows your code to work with the standard `crypt` and with more exotic implementations. In other words, do not assume anything about the returned string itself, or how many bytes in the encrypted string matter.

Traditionally the result is a string of 13 bytes: two first bytes of the salt, followed by 11 bytes from the set `[./0-9A-Za-z]`, and only the first eight bytes of the encrypted string mattered, but alternative hashing schemes (like MD5), higher level security schemes (like C2), and implementations on non-UNIX platforms may produce different strings.

When choosing a new salt create a random two character string whose characters come from the set `[./0-9A-Za-z]` (like `join '', ('.', '/', 0..9, 'A'..'Z', 'a'..'z')` [`rand 64, rand 64`]).

Here's an example that makes sure that whoever runs this program knows their own password:

```
$pwd = (getpwuid($<))[1];
system "stty -echo";
print "Password: ";
chomp($word = <STDIN>);
print "\n";
system "stty echo";

if (crypt($word, $pwd) ne $pwd) {
    die "Sorry...\n";
} else {
    print "ok\n";
}
```

Of course, typing in your own password to whoever asks you for it is unwise.

The `crypt` function is unsuitable for encrypting large quantities of data, not least of all because you can't get the information back. Look at the *by-module/Crypt* and *by-module/PGP* directories on your favorite CPAN mirror for a slew of potentially useful modules.

If using `crypt()` on a Unicode string (which *potentially* has characters with codepoints above 255), Perl tries to make sense of the situation by trying to downgrade (a copy of the string) the string back to an eight-bit byte string before calling `crypt()` (on that copy). If that works, good. If not, `crypt()` dies with `Wide character in crypt`.

dbmclose HASH

[This function has been largely superseded by the `untie` function.]

Breaks the binding between a DBM file and a hash.

dbmopen HASH,DBNAME,MASK

[This function has been largely superseded by the `tie` function.]

This binds a `dbm(3)`, `ndbm(3)`, `sdbm(3)`, `gdbm(3)`, or Berkeley DB file to a hash. `HASH` is the name of the hash. (Unlike normal `open`, the first argument is *not* a filehandle, even though it looks like one). `DBNAME` is the name of the database (without the *.dir* or *.pag* extension if any). If the database does not exist, it is created with protection specified by `MASK` (as modified by the `umask`). If your system supports only the older DBM functions, you may perform only one `dbmopen` in your program. In older versions of Perl, if your system had neither DBM nor `ndbm`, calling `dbmopen` produced a fatal error; it now falls back to `sdbm(3)`.

If you don't have write access to the DBM file, you can only read hash variables, not set them. If you want to test whether you can write, either use file tests or try setting a dummy hash entry inside an `eval`, which will trap the error.

Note that functions such as `keys` and `values` may return huge lists when used on large DBM files. You may prefer to use the `each` function to iterate over large DBM files. Example:

```
# print out history file offsets
dbmopen(%HIST, '/usr/lib/news/history', 0666);
while (($key,$val) = each %HIST) {
```

```
        print $key, ' = ', unpack('L',$val), "\n";
    }
    dbmclose(%HIST);
```

See also [AnyDBM_File](#) for a more general description of the pros and cons of the various dbm approaches, as well as [DB_File](#) for a particularly rich implementation.

You can control which DBM library you use by loading that library before you call `dbmopen()`:

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.db")
    or die "Can't open netscape history file: $!";
```

defined EXPR

defined Returns a Boolean value telling whether EXPR has a value other than the undefined value `undef`. If EXPR is not present, `$_` will be checked.

Many operations return `undef` to indicate failure, end of file, system error, uninitialized variable, and other exceptional conditions. This function allows you to distinguish `undef` from other values. (A simple Boolean test will not distinguish among `undef`, zero, the empty string, and `"0"`, which are all equally false.) Note that since `undef` is a valid scalar, its presence doesn't *necessarily* indicate an exceptional condition: `pop` returns `undef` when its argument is an empty array, *or* when the element to return happens to be `undef`.

You may also use `defined(&func)` to check whether subroutine `&func` has ever been defined. The return value is unaffected by any forward declarations of `&foo`. Note that a subroutine which is not defined may still be callable: its package may have an `AUTOLOAD` method that makes it spring into existence the first time that it is called — see [perlsub](#).

Use of `defined` on aggregates (hashes and arrays) is deprecated. It used to report whether memory for that aggregate has ever been allocated. This behavior may disappear in future versions of Perl. You should instead use a simple test for size:

```
if (@an_array) { print "has array elements\n" }
if (%a_hash)   { print "has hash members\n" }
```

When used on a hash element, it tells you whether the value is defined, not whether the key exists in the hash. Use [/exists](#) for the latter purpose.

Examples:

```
print if defined $switch{'D'};
print "$val\n" while defined($val = pop(@ary));
die "Can't readlink $sym: $!"
    unless defined($value = readlink $sym);
sub foo { defined &$bar ? &$bar(@_) : die "No bar"; }
$debugging = 0 unless defined $debugging;
```

Note: Many folks tend to overuse `defined`, and then are surprised to discover that the number and `" "` (the zero-length string) are, in fact, defined values. For example, if you say

```
"ab" =~ /a(.*?)b/;
```

The pattern match succeeds, and `$1` is defined, despite the fact that it matched "nothing". But it didn't really match nothing—rather, it matched something that happened to be zero characters long. This is all very above-board and honest. When a function returns an undefined value, it's an admission that it couldn't give you an honest answer. So you should use `defined` only when you're questioning the integrity of what you're trying to do. At other times, a simple comparison to `or ""` is what you want.

See also [/undef](#), [/exists](#), [/ref](#).

delete EXPR

Given an expression that specifies a hash element, array element, hash slice, or array slice, deletes the specified element(s) from the hash or array. In the case of an array, if the array elements happen to be at the end, the size of the array will shrink to the highest element that tests true for `exists()` (or 0 if no such element exists).

Returns each element so deleted or the undefined value if there was no such element. Deleting from `$ENV{}` modifies the environment. Deleting from a hash tied to a DBM file deletes the entry from the DBM file. Deleting from a tied hash or array may not necessarily return anything.

Deleting an array element effectively returns that position of the array to its initial, uninitialized state. Subsequently testing for the same element with `exists()` will return false. Note that deleting array elements in the middle of an array will not shift the index of the ones after them down—use `splice()` for that. See [/exists](#).

The following (inefficiently) deletes all the values of `%HASH` and `@ARRAY`:

```
foreach $key (keys %HASH) {
    delete $HASH{$key};
}

foreach $index (0 .. $#ARRAY) {
    delete $ARRAY[$index];
}
```

And so do these:

```
delete @HASH{keys %HASH};

delete @ARRAY[0 .. $#ARRAY];
```

But both of these are slower than just assigning the empty list or undefining `%HASH` or `@ARRAY`:

```
%HASH = ();          # completely empty %HASH
undef %HASH;          # forget %HASH ever existed

@ARRAY = ();          # completely empty @ARRAY
undef @ARRAY;         # forget @ARRAY ever existed
```

Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash element, array element, hash slice, or array slice lookup:

```
delete $ref->[$x][$y]{$key};
delete @{$ref->[$x][$y]}{$key1, $key2, @morekeys};

delete $ref->[$x][$y][$index];
delete @{$ref->[$x][$y]}[$index1, $index2, @moreindices];
```

die LIST Outside an eval, prints the value of LIST to STDERR and exits with the current value of `$!` (errno). If `$!` is , exits with the value of `<< ($? 8)` (backtick 'command' status). If `<< ($? 8)` is , exits with 255. Inside an eval(), the error message is stuffed into `$@` and the eval is terminated with the undefined value. This makes `die` the way to raise an exception.

Equivalent examples:

```
die "Can't cd to spool: $!\n" unless chdir '/usr/spool/news';
chdir '/usr/spool/news' or die "Can't cd to spool: $!\n"
```

If the last element of LIST does not end in a newline, the current script line number and input line number (if any) are also printed, and a newline is supplied. Note that the "input line

number" (also known as "chunk") is subject to whatever notion of "line" happens to be currently in effect, and is also available as the special variable `$.`. See [\\$/ in perlvar](#) and [\\$. in perlvar](#).

Hint: sometimes appending `", stopped"` to your message will cause it to make better sense when the string `"at foo line 123"` is appended. Suppose you are running script `"canasta"`.

```
die "/etc/games is no good";
die "/etc/games is no good, stopped";
```

produce, respectively

```
/etc/games is no good at canasta line 123.
/etc/games is no good, stopped at canasta line 123.
```

See also `exit()`, `warn()`, and the `Carp` module.

If `LIST` is empty and `$@` already contains a value (typically from a previous `eval`) that value is reused after appending `"\t...propagated"`. This is useful for propagating exceptions:

```
eval { ... };
die unless $@ =~ /Expected exception/;
```

If `LIST` is empty and `$@` contains an object reference that has a `PROPAGATE` method, that method will be called with additional file and line number parameters. The return value replaces the value in `$@`. ie. as if `<$@ = eval { $@->PROPAGATE(__FILE__, __LINE__) }`; were called.

If `$@` is empty then the string `"Died"` is used.

`die()` can also be called with a reference argument. If this happens to be trapped within an `eval()`, `$@` contains the reference. This behavior permits a more elaborate exception handling implementation using objects that maintain arbitrary state about the nature of the exception. Such a scheme is sometimes preferable to matching particular string values of `$@` using regular expressions. Here's an example:

```
eval { ... ; die Some::Module::Exception->new( FOO => "bar" ) };
if ($@) {
    if (ref($@) && UNIVERSAL::isa($@, "Some::Module::Exception")) {
        # handle Some::Module::Exception
    }
    else {
        # handle all other possible exceptions
    }
}
```

Because perl will stringify uncaught exception messages before displaying them, you may want to overload stringification operations on such custom exception objects. See [overload](#) for details about that.

You can arrange for a callback to be run just before the `die` does its deed, by setting the `$SIG{__DIE__}` hook. The associated handler will be called with the error text and can change the error message, if it sees fit, by calling `die` again. See [\\$SIG{expr}](#) for details on setting `%SIG` entries, and ["eval BLOCK"](#) for some examples. Although this feature was meant to be run only right before your program was to exit, this is not currently the case—the `$SIG{__DIE__}` hook is currently called even inside `eval()` ed blocks/strings! If one wants the hook to do nothing in such situations, put

```
die @_ if $^S;
```

as the first line of the handler (see [\\$^S](#)). Because this promotes strange action at a distance, this counterintuitive behavior may be fixed in a future release.

do BLOCK

Not really a function. Returns the value of the last command in the sequence of commands indicated by BLOCK. When modified by a loop modifier, executes the BLOCK once before testing the loop condition. (On other statements the loop modifiers test the conditional first.)

`do BLOCK` does *not* count as a loop, so the loop control statements `next`, `last`, or `redo` cannot be used to leave or restart the block. See [perlsyn](#) for alternative strategies.

do SUBROUTINE(LIST)

A deprecated form of subroutine call. See [perlsub](#).

do EXPR Uses the value of EXPR as a filename and executes the contents of the file as a Perl script. Its primary use is to include subroutines from a Perl subroutine library.

```
do 'stat.pl';
```

is just like

```
eval `cat stat.pl`;
```

except that it's more efficient and concise, keeps track of the current filename for error messages, searches the `@INC` libraries, and updates `%INC` if the file is found. See [Predefined Names](#) for these variables. It also differs in that code evaluated with `do FILENAME` cannot see lexicals in the enclosing scope; `eval STRING` does. It's the same, however, in that it does reparse the file every time you call it, so you probably don't want to do this inside a loop.

If `do` cannot read the file, it returns `undef` and sets `$!` to the error. If `do` can read the file but cannot compile it, it returns `undef` and sets an error message in `$@`. If the file is successfully compiled, `do` returns the value of the last expression evaluated.

Note that inclusion of library modules is better done with the `use` and `require` operators, which also do automatic error checking and raise an exception if there's a problem.

You might like to use `do` to read in a program configuration file. Manual error checking can be done this way:

```
# read in config files: system first, then user
for $file ("/share/prog/defaults.rc",
           "$ENV{HOME}/.someprogrc")
{
    unless ($return = do $file) {
        warn "couldn't parse $file: $@" if $@;
        warn "couldn't do $file: $!"   unless defined $return;
        warn "couldn't run $file"      unless $return;
    }
}
```

dump LABEL

dump This function causes an immediate core dump. See also the `-u` command-line switch in [perlrun](#), which does the same thing. Primarily this is so that you can use the **undump** program (not supplied) to turn your core dump into an executable binary after having initialized all your variables at the beginning of the program. When the new binary is executed it will begin by executing a `goto LABEL` (with all the restrictions that `goto` suffers). Think of it as a `goto` with an intervening core dump and reincarnation. If LABEL is omitted, restarts the program from the top.

WARNING: Any files opened at the time of the dump will *not* be open any more when the program is reincarnated, with possible resulting confusion on the part of Perl.

This function is now largely obsolete, partly because it's very hard to convert a core file into an executable, and because the real compiler backends for generating portable bytecode and compilable C code have superseded it. That's why you should now invoke it as `CORE::dump()`, if you don't want to be warned against a possible typo.

If you're looking to use [dump](#) to speed up your program, consider generating bytecode or native C code as described in [perlcc](#). If you're just trying to accelerate a CGI script, consider using the `mod_perl` extension to **Apache**, or the CPAN module, `CGI::Fast`. You might also consider autoloading or selfloading, which at least make your program *appear* to run faster.

each HASH

When called in list context, returns a 2-element list consisting of the key and value for the next element of a hash, so that you can iterate over it. When called in scalar context, returns only the key for the next element in the hash.

Entries are returned in an apparently random order. The actual random order is subject to change in future versions of perl, but it is guaranteed to be in the same order as either the `keys` or `values` function would produce on the same (unmodified) hash.

When the hash is entirely read, a null array is returned in list context (which when assigned produces a false () value), and `undef` in scalar context. The next call to `each` after that will start iterating again. There is a single iterator for each hash, shared by all `each`, `keys`, and `values` function calls in the program; it can be reset by reading all the elements from the hash, or by evaluating `keys HASH` or `values HASH`. If you add or delete elements of a hash while you're iterating over it, you may get entries skipped or duplicated, so don't. Exception: It is always safe to delete the item most recently returned by `each()`, which means that the following code will work:

```
while (($key, $value) = each %hash) {
    print $key, "\n";
    delete $hash{$key};    # This is safe
}
```

The following prints out your environment like the `printenv(1)` program, only in a different order:

```
while (($key,$value) = each %ENV) {
    print "$key=$value\n";
}
```

See also `keys`, `values` and `sort`.

eof FILEHANDLE

`eof ()`

`eof`

Returns 1 if the next read on FILEHANDLE will return end of file, or if FILEHANDLE is not open. FILEHANDLE may be an expression whose value gives the real filehandle. (Note that this function actually reads a character and then `ungetc`s it, so isn't very useful in an interactive context.) Do not read from a terminal file (or call `eof(FILEHANDLE)` on it) after end-of-file is reached. File types such as terminals may lose the end-of-file condition if you do.

An `eof` without an argument uses the last file read. Using `eof()` with empty parentheses is very different. It refers to the pseudo file formed from the files listed on the command line and accessed via the `< <` operator. Since `< <` isn't explicitly opened, as a normal filehandle is, an `eof()` before `< <` has been used will cause `@ARGV` to be examined to determine if input is available. Similarly, an `eof()` after `< <` has returned end-of-file will assume you are processing another `@ARGV` list, and if you haven't set `@ARGV`, will read input from `STDIN`; see [I/O Operators in perlop](#).

In a `< while (<)` loop, `eof` or `eof(ARGV)` can be used to detect the end of each file, `eof()` will only detect the end of the last file. Examples:

```

# reset line numbering on each input file
while (<>) {
    next if /^\\s*#/#; # skip comments
    print "$\\.\\t$\\_";
} continue {
    close ARGV if eof; # eof()!
}

# insert dashes just before last line of last file
while (<>) {
    if (eof()) {                # check for end of current file
        print "-----\\n";
        close(ARGV);           # close or last; is needed if we
                                # are reading from the terminal
    }
    print;
}

```

Practical hint: you almost never need to use `eof` in Perl, because the input operators typically return `undef` when they run out of data, or if there was an error.

eval EXPR

eval BLOCK

In the first form, the return value of `EXPR` is parsed and executed as if it were a little Perl program. The value of the expression (which is itself determined within scalar context) is first parsed, and if there weren't any errors, executed in the lexical context of the current Perl program, so that any variable settings or subroutine and format definitions remain afterwards. Note that the value is parsed every time the `eval` executes. If `EXPR` is omitted, evaluates `$_`. This form is typically used to delay parsing and subsequent execution of the text of `EXPR` until run time.

In the second form, the code within the `BLOCK` is parsed only once—at the same time the code surrounding the `eval` itself was parsed—and executed within the context of the current Perl program. This form is typically used to trap exceptions more efficiently than the first (see below), while also providing the benefit of checking the code within `BLOCK` at compile time.

The final semicolon, if any, may be omitted from the value of `EXPR` or within the `BLOCK`.

In both forms, the value returned is the value of the last expression evaluated inside the mini-program; a return statement may be also used, just as with subroutines. The expression providing the return value is evaluated in void, scalar, or list context, depending on the context of the `eval` itself. See [/wantarray](#) for more on how the evaluation context can be determined.

If there is a syntax error or runtime error, or a `die` statement is executed, an undefined value is returned by `eval`, and `$@` is set to the error message. If there was no error, `$@` is guaranteed to be a null string. Beware that using `eval` neither silences perl from printing warnings to `STDERR`, nor does it stuff the text of warning messages into `$@`. To do either of those, you have to use the `$SIG{__WARN__}` facility, or turn off warnings inside the `BLOCK` or `EXPR` using `no warnings 'all'`. See [/warn](#), [perlvar](#), [warnings](#) and [perllexwarn](#).

Note that, because `eval` traps otherwise-fatal errors, it is useful for determining whether a particular feature (such as `socket` or `symlink`) is implemented. It is also Perl's exception trapping mechanism, where the `die` operator is used to raise exceptions.

If the code to be executed doesn't vary, you may use the `eval-BLOCK` form to trap run-time errors without incurring the penalty of recompiling each time. The error, if any, is still returned in `$@`. Examples:

```
# make divide-by-zero nonfatal
```

```
eval { $answer = $a / $b; }; warn $@ if $@;

# same thing, but less efficient
eval '$answer = $a / $b'; warn $@ if $@;

# a compile-time error
eval { $answer = };                # WRONG

# a run-time error
eval '$answer =';    # sets $@
```

Due to the current arguably broken state of `__DIE__` hooks, when using the `eval{}` form as an exception trap in libraries, you may wish not to trigger any `__DIE__` hooks that user code may have installed. You can use the local `$SIG{__DIE__}` construct for this purpose, as shown in this example:

```
# a very private exception trap for divide-by-zero
eval { local $SIG{'__DIE__'}; $answer = $a / $b; };
warn $@ if $@;
```

This is especially significant, given that `__DIE__` hooks can call `die` again, which has the effect of changing their error messages:

```
# __DIE__ hooks may modify error messages
{
    local $SIG{'__DIE__'} =
        sub { (my $x = $_[0]) =~ s/foo/bar/g; die $x };
    eval { die "foo lives here" };
    print $@ if $@;                # prints "bar lives here"
}
```

Because this promotes action at a distance, this counterintuitive behavior may be fixed in a future release.

With an `eval`, you should be especially careful to remember what's being looked at when:

```
eval $x;                # CASE 1
eval "$x";               # CASE 2

eval '$x';              # CASE 3
eval { $x };            # CASE 4

eval "\$$x++";          # CASE 5
$$x++;                  # CASE 6
```

Cases 1 and 2 above behave identically: they run the code contained in the variable `$x`. (Although case 2 has misleading double quotes making the reader wonder what else might be happening (nothing is).) Cases 3 and 4 likewise behave in the same way: they run the code `'$x'`, which does nothing but return the value of `$x`. (Case 4 is preferred for purely visual reasons, but it also has the advantage of compiling at compile-time instead of at run-time.) Case 5 is a place where normally you *would* like to use double quotes, except that in this particular situation, you can just use symbolic references instead, as in case 6.

`eval BLOCK` does *not* count as a loop, so the loop control statements `next`, `last`, or `redo` cannot be used to leave or restart the block.

exec LIST

exec PROGRAM LIST

The `exec` function executes a system command *and never returns*— use `system` instead of `exec` if you want it to return. It fails and returns false only if the command does not exist *and* it is executed directly instead of via your system's command shell (see below).

Since it's a common mistake to use `exec` instead of `system`, Perl warns you if there is a following statement which isn't `die`, `warn`, or `exit` (if `-w` is set – but you always do that). If you *really* want to follow an `exec` with some other statement, you can use one of these styles to avoid the warning:

```
exec ('foo')    or print STDERR "couldn't exec foo: $!";
{ exec ('foo') }; print STDERR "couldn't exec foo: $!";
```

If there is more than one argument in `LIST`, or if `LIST` is an array with more than one value, calls `execvp(3)` with the arguments in `LIST`. If there is only one scalar argument or an array with one element in it, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is `/bin/sh -c` on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to `execvp`, which is more efficient. Examples:

```
exec '/bin/echo', 'Your arguments are: ', @ARGV;
exec "sort $outfile | uniq";
```

If you don't really want to execute the first argument, but want to lie to the program you are executing about its own name, you can specify the program you actually want to run as an "indirect object" (without a comma) in front of the `LIST`. (This always forces interpretation of the `LIST` as a multivalued list, even if there is only a single scalar in the list.) Example:

```
$shell = '/bin/csh';
exec $shell '-sh';          # pretend it's a login shell
```

or, more directly,

```
exec {'/bin/csh'} '-sh';    # pretend it's a login shell
```

When the arguments get executed via the system shell, results will be subject to its quirks and capabilities. See *'STRING' in perlop* for details.

Using an indirect object with `exec` or `system` is also more secure. This usage (which also works fine with `system()`) forces interpretation of the arguments as a multivalued list, even if the list had just one argument. That way you're safe from the shell expanding wildcards or splitting up words with whitespace in them.

```
@args = ( "echo surprise" );
exec @args;          # subject to shell escapes
                    # if @args == 1
exec { $args[0] } @args; # safe even with one-arg list
```

The first version, the one without the indirect object, ran the `echo` program, passing it "surprise" an argument. The second version didn't—it tried to run a program literally called *"echo surprise"*, didn't find it, and set `$?` to a non-zero value indicating failure.

Beginning with v5.6.0, Perl will attempt to flush all files opened for output before the `exec`, but this may not be supported on some platforms (see *perlport*). To be safe, you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush()` method of `IO::Handle` on any open handles in order to avoid lost output.

Note that `exec` will not call your `END` blocks, nor will it call any `DESTROY` methods in your objects.

exists EXPR

Given an expression that specifies a hash element or array element, returns true if the specified element in the hash or array has ever been initialized, even if the corresponding value is undefined. The element is not autovivified if it doesn't exist.

```
print "Exists\n"    if exists $hash{$key};
```

```

print "Defined\defined $hash{$key};
print "True\n"         if $hash{$key};

print "Exists\n"       if exists $array[$index];
print "Defined\n"      if defined $array[$index];
print "True\n"         if $array[$index];

```

A hash or array element can be true only if it's defined, and defined if it exists, but the reverse doesn't necessarily hold true.

Given an expression that specifies the name of a subroutine, returns true if the specified subroutine has ever been declared, even if it is undefined. Mentioning a subroutine name for exists or defined does not count as declaring it. Note that a subroutine which does not exist may still be callable: its package may have an AUTOLOAD method that makes it spring into existence the first time that it is called — see [perlsub](#).

```

print "Exists\n"       if exists &subroutine;
print "Defined\n"      if defined &subroutine;

```

Note that the EXPR can be arbitrarily complicated as long as the final operation is a hash or array key lookup or subroutine name:

```

if (exists $ref->{A}->{B}->{$key}) { }
if (exists $hash{A}{B}{$key})      { }

if (exists $ref->{A}->{B}->[$ix])   { }
if (exists $hash{A}{B}[$ix])       { }

if (exists &{$ref->{A}{B}{$key}})   { }

```

Although the deepest nested array or hash will not spring into existence just because its existence was tested, any intervening ones will. Thus `< $ref-{ "A" }` and `< $ref-{ "A" }-{ "B" }` will spring into existence due to the existence test for the `$key` element above. This happens anywhere the arrow operator is used, including even:

```

undef $ref;
if (exists $ref->{ "Some key" }) { }
print $ref;                # prints HASH(0x80d3d5c)

```

This surprising autovivification in what does not at first—or even second—glance appear to be an lvalue context may be fixed in a future release.

See [Pseudo-hashes: Using an array as a hash in perlref](#) for specifics on how `exists()` acts when used on a pseudo-hash.

Use of a subroutine call, rather than a subroutine name, as an argument to `exists()` is an error.

```

exists &sub;                # OK
exists &sub();              # Error

```

exit EXPR

Evaluates EXPR and exits immediately with that value. Example:

```

$ans = <STDIN>;
exit 0 if $ans =~ /^[Xx]/;

```

See also `die`. If EXPR is omitted, exits with `status`. The only universally recognized values for EXPR are `0` for success and `1` for error; other values are subject to interpretation depending on the environment in which the Perl program is running. For example, exiting `69` (`EX_UNAVAILABLE`) from a *sendmail* incoming-mail filter will cause the mailer to return the item undelivered, but that's not true everywhere.

Don't use `exit` to abort a subroutine if there's any chance that someone might want to trap whatever error happened. Use `die` instead, which can be trapped by an `eval`.

The `exit()` function does not always exit immediately. It calls any defined `END` routines first, but these `END` routines may not themselves abort the exit. Likewise any object destructors that need to be called are called before the real exit. If this is a problem, you can call `POSIX::_exit($status)` to avoid `END` and destructor processing. See [perlmod](#) for details.

`exp` `EXPR`

`exp` Returns e (the natural logarithm base) to the power of `EXPR`. If `EXPR` is omitted, gives `exp($_)`.

`fcntl` `FILEHANDLE,FUNCTION,SCALAR`

Implements the `fcntl(2)` function. You'll probably have to say

```
use Fcntl;
```

first to get the correct constant definitions. Argument processing and value return works just like `ioctl` below. For example:

```
use Fcntl;
fcntl($filehandle, F_GETFL, $packed_return_buffer)
    or die "can't fcntl F_GETFL: $!";
```

You don't have to check for defined on the return from `fcntl`. Like `ioctl`, it maps a return from the system call into "0 but true" in Perl. This string is true in boolean context and in numeric context. It is also exempt from the normal `-w` warnings on improper numeric conversions.

Note that `fcntl` will produce a fatal error if used on a machine that doesn't implement `fcntl(2)`. See the `Fcntl` module or your `fcntl(2)` manpage to learn what functions are available on your system.

`fileno` `FILEHANDLE`

Returns the file descriptor for a filehandle, or undefined if the filehandle is not open. This is mainly useful for constructing bitmaps for `select` and low-level POSIX tty-handling operations. If `FILEHANDLE` is an expression, the value is taken as an indirect filehandle, generally its name.

You can use this to find out whether two handles refer to the same underlying descriptor:

```
if (fileno(THIS) == fileno(THAT)) {
    print "THIS and THAT are dups\n";
}
```

(Filehandles connected to memory objects via new features of `open` may return undefined even though they are open.)

`flock` `FILEHANDLE,OPERATION`

Calls `flock(2)`, or an emulation of it, on `FILEHANDLE`. Returns true for success, false on failure. Produces a fatal error if used on a machine that doesn't implement `flock(2)`, `fcntl(2)` locking, or `lockf(3)`. `flock` is Perl's portable file locking interface, although it locks only entire files, not records.

Two potentially non-obvious but traditional `flock` semantics are that it waits indefinitely until the lock is granted, and that its locks **merely advisory**. Such discretionary locks are more flexible, but offer fewer guarantees. This means that files locked with `flock` may be modified by programs that do not also use `flock`. See [perlport](#), your port's specific documentation, or your system-specific local manpages for details. It's best to assume traditional behavior if you're writing portable programs. (But if you're not, you should as always feel perfectly free to write for your own system's idiosyncrasies (sometimes called "features"). Slavish adherence to

portability concerns shouldn't get in the way of your getting your job done.)

OPERATION is one of LOCK_SH, LOCK_EX, or LOCK_UN, possibly combined with LOCK_NB. These constants are traditionally valued 1, 2, 8 and 4, but you can use the symbolic names if you import them from the Fcntl module, either individually, or as a group using the `':flock'` tag. LOCK_SH requests a shared lock, LOCK_EX requests an exclusive lock, and LOCK_UN releases a previously requested lock. If LOCK_NB is bitwise-or'ed with LOCK_SH or LOCK_EX then flock will return immediately rather than blocking waiting for the lock (check the return status to see if you got it).

To avoid the possibility of miscoordination, Perl now flushes FILEHANDLE before locking or unlocking it.

Note that the emulation built with lockf(3) doesn't provide shared locks, and it requires that FILEHANDLE be open with write intent. These are the semantics that lockf(3) implements. Most if not all systems implement lockf(3) in terms of fcntl(2) locking, though, so the differing semantics shouldn't bite too many people.

Note that the fcntl(2) emulation of flock(3) requires that FILEHANDLE be open with read intent to use LOCK_SH and requires that it be open with write intent to use LOCK_EX.

Note also that some versions of flock cannot lock things over the network; you would need to use the more system-specific fcntl for that. If you like you can force Perl to ignore your system's flock(2) function, and so provide its own fcntl(2)-based emulation, by passing the switch `-Ud_flock` to the *Configure* program when you configure perl.

Here's a mailbox appender for BSD systems.

```
use Fcntl ':flock'; # import LOCK_* constants

sub lock {
    flock(MBOX, LOCK_EX);
    # and, in case someone appended
    # while we were waiting...
    seek(MBOX, 0, 2);
}

sub unlock {
    flock(MBOX, LOCK_UN);
}

open(MBOX, ">>/usr/spool/mail/$ENV{'USER'}")
    or die "Can't open mailbox: $!";

lock();
print MBOX $msg, "\n\n";
unlock();
```

On systems that support a real flock(), locks are inherited across fork() calls, whereas those that must resort to the more capricious fcntl() function lose the locks, making it harder to write servers.

See also [DB_File](#) for other flock() examples.

fork

Does a fork(2) system call to create a new process running the same program at the same point. It returns the child pid to the parent process, to the child process, or undef if the fork is unsuccessful. File descriptors (and sometimes locks on those descriptors) are shared, while everything else is copied. On most systems supporting fork(), great care has gone into making it extremely efficient (for example, using copy-on-write technology on data pages), making it the dominant paradigm for multitasking over the last few decades.

Beginning with v5.6.0, Perl will attempt to flush all files opened for output before forking the child process, but this may not be supported on some platforms (see [perlport](#)). To be safe, you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush()` method of `IO::Handle` on any open handles in order to avoid duplicate output.

If you fork without ever waiting on your children, you will accumulate zombies. On some systems, you can avoid this by setting `$SIG{CHLD}` to "IGNORE". See also [perlipc](#) for more examples of forking and reaping moribund children.

Note that if your forked child inherits system file descriptors like `STDIN` and `STDOUT` that are actually connected by a pipe or socket, even if you exit, then the remote server (such as, say, a CGI script or a backgrounded job launched from a remote shell) won't think you're done. You should reopen those to `/dev/null` if it's any issue.

format Declare a picture format for use by the `write` function. For example:

```
format Something =
    Test: @<<<<<<<< @||| | @>>>>>
           $str,      $%,      '$' . int($num)
.
$str = "widget";
$num = $cost/$quantity;
$~ = 'Something';
write;
```

See [perlform](#) for many details and examples.

formline PICTURE,LIST

This is an internal function used by formats, though you may call it, too. It formats (see [perlform](#)) a list of values according to the contents of `PICTURE`, placing the output into the format output accumulator, `$_` (or `$ACCUMULATOR` in English). Eventually, when a `write` is done, the contents of `$_` are written to some filehandle, but you could also read `$_` yourself and then set `$_` back to `"`. Note that a format typically does one `formline` per line of form, but the `formline` function itself doesn't care how many newlines are embedded in the `PICTURE`. This means that the `~` and `~~` tokens will treat the entire `PICTURE` as a single line. You may therefore need to use multiple `formlines` to implement a single record format, just like the format compiler.

Be careful if you put double quotes around the picture, because an `@` character may be taken to mean the beginning of an array name. `formline` always returns true. See [perlform](#) for other examples.

getc FILEHANDLE

getc Returns the next character from the input file attached to `FILEHANDLE`, or the undefined value at end of file, or if there was an error. If `FILEHANDLE` is omitted, reads from `STDIN`. This is not particularly efficient. However, it cannot be used by itself to fetch single characters without waiting for the user to hit enter. For that, try something more like:

```
if ($BSD_STYLE) {
    system "stty cbreak </dev/tty >/dev/tty 2>&1";
}
else {
    system "stty", '-icanon', 'eol', "\001";
}

$key = getc(STDIN);

if ($BSD_STYLE) {
    system "stty -cbreak </dev/tty >/dev/tty 2>&1";
```



```

    }
    else {
        system "stty", 'icanon', 'eol', '^@'; # ASCII null
    }
    print "\n";

```

Determination of whether `$BSD_STYLE` should be set is left as an exercise to the reader.

The `POSIX::getattr` function can do this more portably on systems purporting POSIX compliance. See also the `Term::ReadKey` module from your nearest CPAN site; details on CPAN can be found on [CPAN](#).

getlogin Implements the C library function of the same name, which on most systems returns the current login from */etc/utmp*, if any. If null, use `getpwuid`.

```
$login = getlogin || getpwuid($<) || "Kilroy";
```

Do not consider `getlogin` for authentication: it is not as secure as `getpwuid`.

getpeername SOCKET

Returns the packed `sockaddr` address of other end of the `SOCKET` connection.

```

use Socket;
$hersockaddr = getpeername(SOCK);
($port, $iaddr) = sockaddr_in($hersockaddr);
$herhostname = gethostbyaddr($iaddr, AF_INET);
$herstraddr = inet_ntoa($iaddr);

```

getpgrp PID

Returns the current process group for the specified PID. Use a PID of `0` to get the current process group for the current process. Will raise an exception if used on a machine that doesn't implement `getpgrp(2)`. If PID is omitted, returns process group of current process. Note that the POSIX version of `getpgrp` does not accept a PID argument, so only `PID==0` is truly portable.

getppid Returns the process id of the parent process.

getpriority WHICH,WHO

Returns the current priority for a process, a process group, or a user. (See [getpriority\(2\)](#).) Will raise a fatal exception if used on a machine that doesn't implement `getpriority(2)`.

getpwnam NAME

getgrnam NAME

gethostbyname NAME

getnetbyname NAME

getprotobyname NAME

getpwuid UID

getgrgid GID

getservbyname NAME,PROTO

gethostbyaddr ADDR,ADDRTYPE

getnetbyaddr ADDR,ADDRTYPE

getprotobyname NUMBER

getservbyport PORT,PROTO

getpwent

getgrent

gethostent

getnetent

getprotoent

getservent

```

setpwent
setgrent
sethostent STAYOPEN
setnetent STAYOPEN
setprotoent STAYOPEN
setservent STAYOPEN
endpwent
endgrent
endhostent
endnetent
endprotoent
endservent

```

These routines perform the same functions as their counterparts in the system library. In list context, the return values from the various get routines are as follows:

```

($name, $passwd, $uid, $gid,
 $quota, $comment, $gcos, $dir, $shell, $expire) = getpw*
($name, $passwd, $gid, $members) = getgr*
($name, $aliases, $addrtype, $length, @addrs) = gethost*
($name, $aliases, $addrtype, $net) = getnet*
($name, $aliases, $proto) = getproto*
($name, $aliases, $port, $proto) = getserv*

```

(If the entry doesn't exist you get a null list.)

The exact meaning of the `$gcos` field varies but it usually contains the real name of the user (as opposed to the login name) and other information pertaining to the user. Beware, however, that in many system users are able to change this information and therefore it cannot be trusted and therefore the `$gcos` is tainted (see [perlsec](#)). The `$passwd` and `$shell`, user's encrypted password and login shell, are also tainted, because of the same reason.

In scalar context, you get the name, unless the function was a lookup by name, in which case you get the other thing, whatever it is. (If the entry doesn't exist you get the undefined value.) For example:

```

$uid    = getpwnam($name);
$name   = getpwuid($num);
$name   = getpwent();
$gid    = getgrnam($name);
$name   = getgrgid($num);
$name   = getgrent();
#etc.

```

In `getpw*()` the fields `$quota`, `$comment`, and `$expire` are special cases in the sense that in many systems they are unsupported. If the `$quota` is unsupported, it is an empty scalar. If it is supported, it usually encodes the disk quota. If the `$comment` field is unsupported, it is an empty scalar. If it is supported it usually encodes some administrative comment about the user. In some systems the `$quota` field may be `$change` or `$age`, fields that have to do with password aging. In some systems the `$comment` field may be `$class`. The `$expire` field, if present, encodes the expiration period of the account or the password. For the availability and the exact meaning of these fields in your system, please consult your `getpwnam(3)` documentation and your *pwd.h* file. You can also find out from within Perl what your `$quota` and `$comment` fields mean and whether you have the `$expire` field by using the `Config` module and the values `d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, and `d_pwexpire`. Shadow password files are only supported if your vendor has implemented them in the intuitive fashion that calling the regular C library routines gets the shadow versions if you're running under privilege or if there exists the `shadow(3)` functions as found in System V (

this includes Solaris and Linux.) Those systems which implement a proprietary shadow password facility are unlikely to be supported.

The `$members` value returned by `getgr*()` is a space separated list of the login names of the members of the group.

For the `gethost*()` functions, if the `h_errno` variable is supported in C, it will be returned to you via `$?` if the function call fails. The `@addrs` value returned by a successful call is a list of the raw addresses returned by the corresponding system library call. In the Internet domain, each address is four bytes long and you can unpack it by saying something like:

```
($a,$b,$c,$d) = unpack('C4',$addr[0]);
```

The Socket library makes this slightly easier:

```
use Socket;
$iaddr = inet_aton("127.1"); # or whatever address
$name  = gethostbyaddr($iaddr, AF_INET);

# or going the other way
$straddr = inet_ntoa($iaddr);
```

If you get tired of remembering which element of the return list contains which return value, by-name interfaces are provided in standard modules: `File::stat`, `Net::hostent`, `Net::netent`, `Net::protoent`, `Net::servent`, `Time::gmtime`, `Time::localtime`, and `User::grent`. These override the normal built-ins, supplying versions that return objects with the appropriate names for each field. For example:

```
use File::stat;
use User::pwent;
$is_his = (stat($filename)->uid == pwent($whoever)->uid);
```

Even though it looks like they're the same method calls (`uid`), they aren't, because a `File::stat` object is different from a `User::pwent` object.

getsockname SOCKET

Returns the packed `sockaddr` address of this end of the `SOCKET` connection, in case you don't know the address because you have several different IPs that the connection might have come in on.

```
use Socket;
$mysockaddr = getsockname(SOCK);
($port, $myaddr) = sockaddr_in($mysockaddr);
printf "Connect to %s [%s]\n",
    scalar gethostbyaddr($myaddr, AF_INET),
    inet_ntoa($myaddr);
```

getsockopt SOCKET,LEVEL,OPTNAME

Returns the socket option requested, or `undef` if there is an error.

glob EXPR

glob In list context, returns a (possibly empty) list of filename expansions on the value of `EXPR` such as the standard Unix shell `/bin/csh` would do. In scalar context, `glob` iterates through such filename expansions, returning `undef` when the list is exhausted. This is the internal function implementing the `< <*.c` operator, but you can use it directly. If `EXPR` is omitted, `$_` is used. The `< <*.c` operator is discussed in more detail in [I/O Operators in *perlop*](#).

Beginning with v5.6.0, this operator is implemented using the standard `File::Glob` extension. See [File::Glob](#) for details.

gmtime EXPR

Converts a time as returned by the time function to an 8–element list with the time localized for the standard Greenwich time zone. Typically used as follows:

```
# 0      1      2      3      4      5      6      7
($sec, $min, $hour, $mday, $mon, $year, $yday, $yday) =
    gmtime(time);
```

All list elements are numeric, and come straight out of the C ‘struct tm’. `$sec`, `$min`, and `$hour` are the seconds, minutes, and hours of the specified time. `$mday` is the day of the month, and `$mon` is the month itself, in the range 0..11 with 0 indicating January and 11 indicating December. `$year` is the number of years since 1900. That is, `$year` is 123 in year 2023. `$yday` is the day of the year, in the range 0..364 (or 0..365 in leap years.)

Note that the `$year` element is *not* simply the last two digits of the year. If you assume it is, then you create non–Y2K–compliant programs—and you wouldn’t want to do that, would you?

The proper way to get a complete 4–digit year is simply:

```
$year += 1900;
```

And to get the last two digits of the year (e.g., ‘01’ in 2001) do:

```
$year = sprintf("%02d", $year % 100);
```

If `EXPR` is omitted, `gmtime()` uses the current time (`gmtime(time)`).

In scalar context, `gmtime()` returns the `ctime(3)` value:

```
$now_string = gmtime; # e.g., "Thu Oct 13 04:54:34 1994"
```

Also see the `timegm` function provided by the `Time::Local` module, and the `strftime(3)` function available via the `POSIX` module.

This scalar value is **not** locale dependent (see [perllocale](#)), but is instead a Perl builtin. Also see the `Time::Local` module, and the `strftime(3)` and `mktime(3)` functions available via the `POSIX` module. To get somewhat similar but locale dependent date strings, set up your locale environment variables appropriately (please see [perllocale](#)) and try for example:

```
use POSIX qw(strftime);
$now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

Note that the `%a` and `%b` escapes, which represent the short forms of the day of the week and the month of the year, may not necessarily be three characters wide in all locales.

goto LABEL**goto EXPR****goto &NAME**

The `goto–LABEL` form finds the statement labeled with `LABEL` and resumes execution there. It may not be used to go into any construct that requires initialization, such as a subroutine or a `foreach` loop. It also can’t be used to go into a construct that is optimized away, or to get out of a block or subroutine given to `sort`. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it’s usually better to use some other construct such as `last` or `die`. The author of Perl has never felt the need to use this form of `goto` (in Perl, that is—C is another matter). (The difference being that C does not offer named loops combined with loop control. Perl does, and this replaces most structured uses of `goto` in other languages.)

The `goto–EXPR` form expects a label name, whose scope will be resolved dynamically. This allows for computed `gotos` per FORTRAN, but isn’t necessarily recommended if you’re optimizing for maintainability:

```
goto ("FOO", "BAR", "GLARCH")[$i];
```

The `goto-&NAME` form is quite different from the other forms of `goto`. In fact, it isn't a `goto` in the normal sense at all, and doesn't have the stigma associated with other `gotos`. Instead, it exits the current subroutine (losing any changes set by `local()`) and immediately calls in its place the named subroutine using the current value of `@_`. This is used by `AUTOLOAD` subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to `@_` in the current subroutine are propagated to the other subroutine.) After the `goto`, not even `caller` will be able to tell that this routine was called first.

`NAME` needn't be the name of a subroutine; it can be a scalar variable containing a code reference, or a block which evaluates to a code reference.

grep BLOCK LIST

grep EXPR,LIST

This is similar in spirit to, but not the same as, `grep(1)` and its relatives. In particular, it is not limited to using regular expressions.

Evaluates the `BLOCK` or `EXPR` for each element of `LIST` (locally setting `$_` to each element) and returns the list value consisting of those elements for which the expression evaluated to true. In scalar context, returns the number of times the expression was true.

```
@foo = grep(!/^#/ , @bar);    # weed out comments
```

or equivalently,

```
@foo = grep {!/^#/} @bar;    # weed out comments
```

Note that `$_` is an alias to the list value, so it can be used to modify the elements of the `LIST`. While this is useful and supported, it can cause bizarre results if the elements of `LIST` are not variables. Similarly, `grep` returns aliases into the original list, much as a `for` loop's index variable aliases the list elements. That is, modifying an element of a list returned by `grep` (for example, in a `foreach`, `map` or another `grep`) actually modifies the element in the original list. This is usually something to be avoided when writing clear code.

See also [/map](#) for a list composed of the results of the `BLOCK` or `EXPR`.

hex EXPR

hex Interprets `EXPR` as a hex string and returns the corresponding value. (To convert strings that might start with either `0`, `0x`, or `0b`, see [/oct](#).) If `EXPR` is omitted, uses `$_`.

```
print hex '0xAf'; # prints '175'
print hex 'aF';   # same
```

Hex strings may only represent integers. Strings that would cause integer overflow trigger a warning. Leading whitespace is not stripped, unlike `oct()`.

import There is no builtin `import` function. It is just an ordinary method (subroutine) defined (or inherited) by modules that wish to export names to another module. The `use` function calls the `import` method for the package used. See also [/use](#), [perlmod](#), and [Exporter](#).

index STR,SUBSTR,POSITION

index STR,SUBSTR

The `index` function searches for one string within another, but without the wildcard-like behavior of a full regular-expression pattern match. It returns the position of the first occurrence of `SUBSTR` in `STR` at or after `POSITION`. If `POSITION` is omitted, starts searching from the beginning of the string. The return value is based at (or whatever you've set the `$[` variable to—but don't do that). If the substring is not found, returns one less than the base, ordinarily `-1`.

int EXPR

int Returns the integer portion of EXPR. If EXPR is omitted, uses `$_`. You should not use this function for rounding: one because it truncates towards 0, and two because machine representations of floating point numbers can sometimes produce counterintuitive results. For example, `int(-6.725/0.025)` produces -268 rather than the correct -269; that's because it's really more like -268.99999999999994315658 instead. Usually, the `sprintf`, `printf`, or the `POSIX::floor` and `POSIX::ceil` functions will serve you better than `int()`.

ioctl FILEHANDLE,FUNCTION,SCALAR

Implements the `ioctl(2)` function. You'll probably first have to say

```
require "ioctl.ph"; # probably in /usr/local/lib/perl/ioctl.ph
```

to get the correct function definitions. If *ioctl.ph* doesn't exist or doesn't have the correct definitions you'll have to roll your own, based on your C header files such as `<sys/ioctl.h`. (There is a Perl script called **h2ph** that comes with the Perl kit that may help you in this, but it's nontrivial.) SCALAR will be read and/or written depending on the FUNCTION—a pointer to the string value of SCALAR will be passed as the third argument of the actual `ioctl` call. (If SCALAR has no string value but does have a numeric value, that value will be passed rather than a pointer to the string value. To guarantee this to be true, add a `to` to the scalar before using it.) The `pack` and `unpack` functions may be needed to manipulate the values of structures used by `ioctl`.

The return value of `ioctl` (and `fcntl`) is as follows:

if OS returns:	then Perl returns:
-1	undefined value
0	string "0 but true"
anything else	that number

Thus Perl returns true on success and false on failure, yet you can still easily determine the actual value returned by the operating system:

```
$retval = ioctl(...) || -1;
printf "System returned %d\n", $retval;
```

The special string "0 but true" is exempt from `-w` complaints about improper numeric conversions.

Here's an example of setting a filehandle named REMOTE to be non-blocking at the system level. You'll have to negotiate `$|` on your own, though.

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

$flags = fcntl(REMOTE, F_GETFL, 0)
    or die "Can't get flags for the socket: $!\n";

$flags = fcntl(REMOTE, F_SETFL, $flags | O_NONBLOCK)
    or die "Can't set flags for the socket: $!\n";
```

join EXPR,LIST

Joins the separate strings of LIST into a single string with fields separated by the value of EXPR, and returns that new string. Example:

```
$rec = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);
```

Beware that unlike `split`, `join` doesn't take a pattern as its first argument. Compare [/split](#).

keys HASH

Returns a list consisting of all the keys of the named hash. (In scalar context, returns the number of keys.) The keys are returned in an apparently random order. The actual random order is

subject to change in future versions of perl, but it is guaranteed to be the same order as either the values or each function produces (given that the hash has not been modified). As a side effect, it resets HASH's iterator.

Here is yet another way to print your environment:

```
@keys = keys %ENV;
@values = values %ENV;
while (@keys) {
    print pop(@keys), '=', pop(@values), "\n";
}
```

or how about sorted by key:

```
foreach $key (sort(keys %ENV)) {
    print $key, '=', $ENV{$key}, "\n";
}
```

The returned values are copies of the original keys in the hash, so modifying them will not affect the original hash. Compare [/values](#).

To sort a hash by value, you'll need to use a `sort` function. Here's a descending numeric sort of a hash by its values:

```
foreach $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
    printf "%4d %s\n", $hash{$key}, $key;
}
```

As an lvalue `keys` allows you to increase the number of hash buckets allocated for the given hash. This can gain you a measure of efficiency if you know the hash is going to get big. (This is similar to pre-extending an array by assigning a larger number to `$#array`.) If you say

```
keys %hash = 200;
```

then `%hash` will have at least 200 buckets allocated for it—256 of them, in fact, since it rounds up to the next power of two. These buckets will be retained even if you do `%hash = ()`, use `undef %hash` if you want to free the storage while `%hash` is still in scope. You can't shrink the number of buckets allocated for the hash using `keys` in this way (but you needn't worry about doing this by accident, as trying has no effect).

See also `each`, `values` and `sort`.

kill SIGNAL, LIST

Sends a signal to a list of processes. Returns the number of processes successfully signaled (which is not necessarily the same as the number actually killed).

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
```

If `SIGNAL` is zero, no signal is sent to the process. This is a useful way to check that the process is alive and hasn't changed its UID. See [perlport](#) for notes on the portability of this construct.

Unlike in the shell, if `SIGNAL` is negative, it kills process groups instead of processes. (On System V, a negative *PROCESS* number will also kill process groups, but that's not portable.) That means you usually want to use positive not negative signals. You may also use a signal name in quotes. See [Signals in perlipc](#) for details.

last LABEL

The `last` command is like the `break` statement in C (as used in loops); it immediately exits the loop in question. If the `LABEL` is omitted, the command refers to the innermost enclosing loop. The `continue` block, if any, is not executed:

```

LINE: while (<STDIN>) {
    last LINE if /^$#;exit when done with header
    #...
}

```

`last` cannot be used to exit a block which returns a value such as `eval { }`, `sub { }` or `do { }`, and should not be used to exit a `grep()` or `map()` operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus `last` can be used to effect an early exit out of such a block.

See also [/continue](#) for an illustration of how `last`, `next`, and `redo` work.

lc EXPR

lc Returns a lowercased version of EXPR. This is the internal function implementing the `\L` escape in double-quoted strings. Respects current `LC_CTYPE` locale if `use locale` in force. See [perllocale](#) and [perlunicode](#) for more details about locale and Unicode support.

If EXPR is omitted, uses `$_`.

lcfirst EXPR

lcfirst Returns the value of EXPR with the first character lowercased. This is the internal function implementing the `\l` escape in double-quoted strings. Respects current `LC_CTYPE` locale if `use locale` in force. See [perllocale](#) and [perlunicode](#) for more details about locale and Unicode support.

If EXPR is omitted, uses `$_`.

length EXPR

length Returns the length in characters of the value of EXPR. If EXPR is omitted, returns length of `$_`. Note that this cannot be used on an entire array or hash to find out how many elements these have. For that, use `scalar @array` and `scalar keys %hash` respectively.

link OLDFILE,NEWFILE

Creates a new filename linked to the old filename. Returns true for success, false otherwise.

listen SOCKET,QUEUESIZE

Does the same thing that the `listen` system call does. Returns true if it succeeded, false otherwise. See the example in [Sockets: Client/Server Communication in perlipc](#).

local EXPR

You really probably want to be using `my` instead, because `local` isn't what most people think of as "local". See ["Private Variables via my \(\)"](#) for details.

A `local` modifies the listed variables to be local to the enclosing block, file, or `eval`. If more than one value is listed, the list must be placed in parentheses. See

["Temporary Values via local \(\)"](#) for details, including issues with tied arrays and hashes.

localtime EXPR

Converts a time as returned by the `time` function to a 9-element list with the time analyzed for the local time zone. Typically used as follows:

```

# 0      1      2      3      4      5      6      7      8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
    localtime(time);

```

All list elements are numeric, and come straight out of the C 'struct tm'. `$sec`, `$min`, and `$hour` are the seconds, minutes, and hours of the specified time. `$mday` is the day of the month, and `$mon` is the month itself, in the range 0..11 with 0 indicating January and 11 indicating December. `$year` is the number of years since 1900. That is, `$year` is 123 in year 2023. `$wday` is the day of the week, with 0 indicating Sunday and 3 indicating Wednesday.

`$yday` is the day of the year, in the range 0..364 (or 0..365 in leap years.) `$isdst` is true if the specified time occurs during daylight savings time, false otherwise.

Note that the `$year` element is *not* simply the last two digits of the year. If you assume it is, then you create non-Y2K-compliant programs—and you wouldn't want to do that, would you?

The proper way to get a complete 4-digit year is simply:

```
$year += 1900;
```

And to get the last two digits of the year (e.g., '01' in 2001) do:

```
$year = sprintf("%02d", $year % 100);
```

If `EXPR` is omitted, `localtime()` uses the current time (`localtime(time)`).

In scalar context, `localtime()` returns the `ctime(3)` value:

```
$now_string = localtime; # e.g., "Thu Oct 13 04:54:34 1994"
```

This scalar value is **not** locale dependent, see [perllocale](#), but instead a Perl builtin. Also see the `Time::Local` module (to convert the second, minutes, hours, ... back to seconds since the stroke of midnight the 1st of January 1970, the value returned by `time()`), and the `strftime(3)` and `mktime(3)` functions available via the `POSIX` module. To get somewhat similar but locale dependent date strings, set up your locale environment variables appropriately (please see [perllocale](#)) and try for example:

```
use POSIX qw(strftime);
$now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
```

Note that the `%a` and `%b`, the short forms of the day of the week and the month of the year, may not necessarily be three characters wide.

lock *THING*

This function places an advisory lock on a shared variable, or referenced object contained in *THING* until the lock goes out of scope.

`lock()` is a "weak keyword": this means that if you've defined a function by this name (before any calls to it), that function will be called instead. (However, if you've said `use threads`, `lock()` is always a keyword.) See [threads](#).

log *EXPR*

Returns the natural logarithm (base *e*) of *EXPR*. If *EXPR* is omitted, returns log of `$_`. To get the log of another base, use basic algebra: The base-*N* log of a number is equal to the natural log of that number divided by the natural log of *N*. For example:

```
sub log10 {
    my $n = shift;
    return log($n)/log(10);
}
```

See also [/exp](#) for the inverse operation.

lstat *EXPR*

Does the same thing as the `stat` function (including setting the special `_filehandle`) but stats a symbolic link instead of the file the symbolic link points to. If symbolic links are unimplemented on your system, a normal `stat` is done.

If *EXPR* is omitted, stats `$_`.

`m//` The match operator. See [perlop](#).

map BLOCK LIST

map EXPR,LIST

Evaluates the BLOCK or EXPR for each element of LIST (locally setting `$_` to each element) and returns the list value composed of the results of each such evaluation. In scalar context, returns the total number of elements so generated. Evaluates BLOCK or EXPR in list context, so each element of LIST may produce zero, one, or more elements in the returned value.

```
@chars = map(chr, @nums);
```

translates a list of numbers to the corresponding characters. And

```
%hash = map { getkey($_) => $_ } @array;
```

is just a funny way to write

```
%hash = ();
foreach $_ (@array) {
    $hash{getkey($_)} = $_;
}
```

Note that `$_` is an alias to the list value, so it can be used to modify the elements of the LIST. While this is useful and supported, it can cause bizarre results if the elements of LIST are not variables. Using a regular `foreach` loop for this purpose would be clearer in most cases. See also [/grep](#) for an array composed of those items of the original list for which the BLOCK or EXPR evaluates to true.

`{` starts both hash references and blocks, so `map { ...` could be either the start of `map BLOCK LIST` or `map EXPR, LIST`. Because perl doesn't look ahead for the closing `}` it has to take a guess at which it's dealing with based on what it finds just after the `{`. Usually it gets it right, but if it doesn't it won't realize something is wrong until it gets to the `}` and encounters the missing (or unexpected) comma. The syntax error will be reported close to the `}` but you'll need to change something near the `{` such as using a unary `+` to give perl some help:

```
%hash = map { "\L$_", 1 } @array # perl guesses EXPR. wrong
%hash = map { +"\L$_", 1 } @array # perl guesses BLOCK. right
%hash = map { ("L$_", 1) } @array # this also works
%hash = map { lc($_), 1 } @array # as does this.
%hash = map +( lc($_), 1 ), @array # this is EXPR and works!

%hash = map ( lc($_), 1 ), @array # evaluates to (1, @array)
```

or to force an anon hash constructor use `+`

```
@hashes = map +{ lc($_), 1 }, @array # EXPR, so needs , at end
```

and you get list of anonymous hashes each with only 1 entry.

mkdir FILENAME,MASK

mkdir FILENAME

Creates the directory specified by FILENAME, with permissions specified by MASK (as modified by `umask`). If it succeeds it returns true, otherwise it returns false and sets `$!` (`errno`). If omitted, MASK defaults to 0777.

In general, it is better to create directories with permissive MASK, and let the user modify that with their `umask`, than it is to supply a restrictive MASK and give the user no way to be more permissive. The exceptions to this rule are when the file or directory should be kept private (mail files, for instance). The `perlfunc(1)` entry on `umask` discusses the choice of MASK in more detail.

Note that according to the POSIX 1003.1-1996 the FILENAME may have any number of trailing slashes. Some operating and filesystems do not get this right, so Perl automatically

removes all trailing slashes to keep everyone happy.

msgctl ID,CMD,ARG

Calls the System V IPC function `msgctl(2)`. You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If `CMD` is `IPC_STAT`, then `ARG` must be a variable which will hold the returned `msqid_ds` structure. Returns like `ioctl`: the undefined value for error, "0 but true" for zero, or the actual return value otherwise. See also [SysV IPC in *perlipc*](#), `IPC::SysV`, and `IPC::Semaphore` documentation.

msgget KEY,FLAGS

Calls the System V IPC function `msgget(2)`. Returns the message queue id, or the undefined value if there is an error. See also [SysV IPC in *perlipc*](#) and `IPC::SysV` and `IPC::Msg` documentation.

msgrcv ID,VAR,SIZE,TYPE,FLAGS

Calls the System V IPC function `msgrcv` to receive a message from message queue `ID` into variable `VAR` with a maximum message size of `SIZE`. Note that when a message is received, the message type as a native long integer will be the first thing in `VAR`, followed by the actual message. This packing may be opened with `unpack("l! a*")`. Taints the variable. Returns true if successful, or false if there is an error. See also [SysV IPC in *perlipc*](#), `IPC::SysV`, and `IPC::SysV::Msg` documentation.

msgsnd ID,MSG,FLAGS

Calls the System V IPC function `msgsnd` to send the message `MSG` to the message queue `ID`. `MSG` must begin with the native long integer message type, and be followed by the length of the actual message, and finally the message itself. This kind of packing can be achieved with `pack("l! a*", $type, $message)`. Returns true if successful, or false if there is an error. See also `IPC::SysV` and `IPC::SysV::Msg` documentation.

my EXPR

my TYPE EXPR

my EXPR : ATTRS

my TYPE EXPR : ATTRS

A `my` declares the listed variables to be local (lexically) to the enclosing block, file, or `eval`. If more than one value is listed, the list must be placed in parentheses.

The exact semantics and interface of `TYPE` and `ATTRS` are still evolving. `TYPE` is currently bound to the use of `fields` pragma, and attributes are handled using the `attributes` pragma, or starting from Perl 5.8.0 also via the `Attribute::Handlers` module. See ["Private Variables via `my\(\)`"](#) for details, and [fields](#), [attributes](#), and [Attribute::Handlers](#).

next LABEL

`next` The next command is like the `continue` statement in C; it starts the next iteration of the loop:

```
LINE: while (<STDIN>) {
    next LINE if /^#/;      # discard comments
    #...
}
```

Note that if there were a `continue` block on the above, it would get executed even on discarded lines. If the `LABEL` is omitted, the command refers to the innermost enclosing loop.

`next` cannot be used to exit a block which returns a value such as `eval { }`, `sub { }` or `do { }`, and should not be used to exit a `grep()` or `map()` operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus `next` will

exit such a block early.

See also [/continue](#) for an illustration of how `last`, `next`, and `redo` work.

`no Module VERSION LIST`
`no Module VERSION`
`no Module LIST`
`no Module`

See the [/use](#) function, which `no` is the opposite of.

`oct EXPR`

`oct` Interprets `EXPR` as an octal string and returns the corresponding value. (If `EXPR` happens to start off with `0x`, interprets it as a hex string. If `EXPR` starts off with `0b`, it is interpreted as a binary string. Leading whitespace is ignored in all three cases.) The following will handle decimal, binary, octal, and hex in the standard Perl or C notation:

```
$val = oct($val) if $val =~ /^0/;
```

If `EXPR` is omitted, uses `$_`. To go the other way (produce a number in octal), use `sprintf()` or `printf()`:

```
$perms = (stat("filename"))[2] & 07777;  
$oct_perms = sprintf "%lo", $perms;
```

The `oct()` function is commonly used when a string such as `644` needs to be converted into a file mode, for example. (Although perl will automatically convert strings into numbers as needed, this automatic conversion assumes base 10.)

`open FILEHANDLE,EXPR`
`open FILEHANDLE,MODE,EXPR`
`open FILEHANDLE,MODE,EXPR,LIST`
`open FILEHANDLE,MODE,REFERENCE`
`open FILEHANDLE`

Opens the file whose filename is given by `EXPR`, and associates it with `FILEHANDLE`.

(The following is a comprehensive reference to `open()`: for a gentler introduction you may consider [perlopentut](#).)

If `FILEHANDLE` is an undefined lexical (`my`) variable the variable is assigned a reference to a new anonymous filehandle, otherwise if `FILEHANDLE` is an expression, its value is used as the name of the real filehandle wanted. (This is considered a symbolic reference, so use `strict 'refs'` should *not* be in effect.)

If `EXPR` is omitted, the scalar variable of the same name as the `FILEHANDLE` contains the filename. (Note that lexical variables—those declared with `my`—will not work for this purpose; so if you're using `my`, specify `EXPR` in your call to `open`.)

If three or more arguments are specified then the mode of opening and the file name are separate. If `MODE` is `< '<'` or nothing, the file is opened for input. If `MODE` is `< ''`, the file is truncated and opened for output, being created if necessary. If `MODE` is `<< ''`, the file is opened for appending, again being created if necessary.

You can put a `'+'` in front of the `< ''` or `< '<'` to indicate that you want both read and write access to the file; thus `< '+<'` is almost always preferred for read/write updates—the `< '+'` mode would clobber the file first. You can't usually use either read-write mode for updating textfiles, since they have variable length records. See the `-i` switch in [perlrun](#) for a better approach. The file is created with permissions of `0666` modified by the process' `umask` value.

These various prefixes correspond to the `fopen(3)` modes of `'r'`, `'r+'`, `'w'`, `'w+'`, `'a'`, and `'a+'`.

In the 2-arguments (and 1-argument) form of the call the mode and filename should be concatenated (in this order), possibly separated by spaces. It is possible to omit the mode in these forms if the mode is < ' < ' .

If the filename begins with a `'|'`, the filename is interpreted as a command to which output is to be piped, and if the filename ends with a `'|'`, the filename is interpreted as a command which pipes output to us. See *"Using open() for IPC"* for more examples of this. (You are not allowed to open to a command that pipes both in *and* out, but see *IPC::Open2*, *IPC::Open3*, and *Bidirectional Communication with Another Process in perlipc* for alternatives.)

For three or more arguments if `MODE` is `'|'` or `'-'`, the filename is interpreted as a command to which output is to be piped, and if `MODE` is `'|'` or `'>|'`, the filename is interpreted as a command which pipes output to us. In the 2-arguments (and 1-argument) form one should replace dash (`'-'`) with the command. See *"Using `open()` for IPC"* for more examples of this. (You are not allowed to `open` to a command that pipes both in *and* out, but see *`IPC::Open2`*, *`IPC::Open3`*, and *`Bidirectional Communication in perlipc`* for alternatives.)

In the three-or-more argument form of pipe opens, if LIST is specified (extra arguments after the command name) then LIST becomes arguments to the command invoked if the platform supports it. The meaning of open with more than three arguments for non-pipe modes is not yet specified. Experimental "layers" may give extra LIST arguments meaning.

In the 2-arguments (and 1-argument) form opening `'-'` opens STDIN and opening `< '-'` opens STDOUT.

You may use the three-argument form of `open` to specify IO "layers" (sometimes also referred to as "disciplines") to be applied to the handle that affect how the input and output are processed (see *open* and *PerlIO* for more details). For example

```
open(FH, "<:utf8", "file")
```

will open the UTF-8 encoded file containing Unicode characters, see [perluniintro](#). (Note that if layers are specified in the three-arg form then default layers set by the `open` pragma are ignored.)

Open returns nonzero upon success, the undefined value otherwise. If the `open` involved a pipe, the return value happens to be the pid of the subprocess.

If you're running Perl on a system that distinguishes between text files and binary files, then you should check out [/binmode](#) for tips for dealing with this. The key distinction between systems that need `binmode` and those that don't is their text file formats. Systems like Unix, Mac OS, and Plan 9, which delimit lines with a single character, and which encode that character in C as `"\n"`, do not need `binmode`. The rest need it.

When opening a file, it's usually a bad idea to continue normal execution if the request failed, so `open` is frequently used in connection with `die`. Even if `die` won't do what you want (say, in a CGI script, where you want to make a nicely formatted error message (but there are modules that can help with that problem)) you should always check the return value from opening a file. The infrequent exception is when working with an unopened filehandle is actually what you want to do.

As a special case the 3 arg form with a read/write mode and the third argument being undef:

```
open(TMP, "+>", undef) or die ...
```

opens a filehandle to an anonymous temporary file.

File handles can be opened to "in memory" files held in Perl scalars via:

```
open($fh, '>', \ $variable) || ..
```

Though if you try to re-open STDOUT or STDERR as an "in memory" file, you have to close it

first:

```
close STDOUT;
open STDOUT, '>', \ $variable or die "Can't open STDOUT: $!";
```

Examples:

```
$ARTICLE = 100;
open ARTICLE or die "Can't find article $ARTICLE: $!\n";
while (<ARTICLE>) { ...

open(LOG, '>>/usr/spool/news/twitlog');      # (log is reserved)
# if the open fails, output is discarded

open(DBASE, '+<', 'dbase.mine')              # open for update
or die "Can't open 'dbase.mine' for update: $!";

open(DBASE, '+<dbase.mine')                  # ditto
or die "Can't open 'dbase.mine' for update: $!";

open(ARTICLE, '-|', "caesar <$article")      # decrypt article
or die "Can't start caesar: $!";

open(ARTICLE, "caesar <$article |")          # ditto
or die "Can't start caesar: $!";

open(EXTRACT, "|sort >/tmp/Tmp$$")           # $$ is our process id
or die "Can't start sort: $!";

# in memory files
open(MEMORY, '>', \ $var)
or die "Can't open memory file: $!";
print MEMORY "foo!\n";                      # output will end up in $var

# process argument list of files along with any includes
foreach $file (@ARGV) {
    process($file, 'fh00');
}

sub process {
    my($filename, $input) = @_;
    $input++;                               # this is a string increment
    unless (open($input, $filename)) {
        print STDERR "Can't open $filename: $!\n";
        return;
    }

    local $_;
    while (<$input>) {                      # note use of indirection
        if (/^#include "(.*)"/) {
            process($1, $input);
            next;
        }
        #...                               # whatever
    }
}
```

You may also, in the Bourne shell tradition, specify an EXPR beginning with `< '&'`, in which case the rest of the string is interpreted as the name of a filehandle (or file descriptor, if numeric) to be duped and opened. You may use `&` after `<`, `<<`, `< <`, `< +`, `<< +`, and `< +<`. The mode you specify should match the mode of the original filehandle. (Duping a filehandle does

not take into account any existing contents of IO buffers.) If you use the 3 arg form then you can pass either a number, the name of a filehandle or the normal "reference to a glob".

Here is a script that saves, redirects, and restores STDOUT and STDERR using various methods:

```
#!/usr/bin/perl
open my $oldout, ">&STDOUT"      or die "Can't dup STDOUT: $!";
open OLDERR,      ">&", \*STDERR or die "Can't dup STDERR: $!";

open STDOUT, '>', "foo.out" or die "Can't redirect STDOUT: $!";
open STDERR, ">&STDOUT"      or die "Can't dup STDOUT: $!";

select STDERR; $| = 1;      # make unbuffered
select STDOUT; $| = 1;      # make unbuffered

print STDOUT "stdout 1\n"; # this works for
print STDERR "stderr 1\n"; # subprocesses too

close STDOUT;
close STDERR;

open STDOUT, ">&", $oldout or die "Can't dup \$oldout: $!";
open STDERR, ">&OLDERR"     or die "Can't dup OLDERR: $!";

print STDOUT "stdout 2\n";
print STDERR "stderr 2\n";
```

If you specify `< '<&=N'`, where N is a number, then Perl will do an equivalent of C's `fdopen` of that file descriptor; this is more parsimonious of file descriptors. For example:

```
open(FILEHANDLE, "<&=$fd")
```

or

```
open(FILEHANDLE, "<&=", $fd)
```

Note that if Perl is using the standard C libraries' `fdopen()` then on many UNIX systems, `fdopen()` is known to fail when file descriptors exceed a certain value, typically 255. If you need more file descriptors than that, consider rebuilding Perl to use the `PerlIO`.

You can see whether Perl has been compiled with `PerlIO` or not by running `perl -V` and looking for `useperlio=` line. If `useperlio` is define, you have `PerlIO`, otherwise you don't.

If you open a pipe on the command `'-'`, i.e., either `'| -'` or `'-|'` with 2-arguments (or 1-argument) form of `open()`, then there is an implicit fork done, and the return value of `open` is the pid of the child within the parent process, and within the child process. (Use `defined($pid)` to determine whether the open was successful.) The filehandle behaves normally for the parent, but i/o to that filehandle is piped from/to the `STDOUT/STDIN` of the child process. In the child process the filehandle isn't opened—i/o happens from/to the new `STDOUT` or `STDIN`. Typically this is used like the normal piped open when you want to exercise more control over just how the pipe command gets executed, such as when you are running `setuid`, and don't want to have to scan shell commands for metacharacters. The following triples are more or less equivalent:

```
open(FOO, "|tr '[a-z]' '[A-Z]')";
open(FOO, '|-', "tr '[a-z]' '[A-Z]')";
open(FOO, '|-') || exec 'tr', '[a-z]', '[A-Z]';
open(FOO, '|-', "tr", '[a-z]', '[A-Z]');

open(FOO, "cat -n '$file'|");
open(FOO, '-|', "cat -n '$file'");
open(FOO, '-|') || exec 'cat', '-n', $file;
```

```
open(FOO, '-|', "cat", '-n', $file);
```

The last example in each block shows the pipe as "list form", which is not yet supported on all platforms. A good rule of thumb is that if your platform has true `fork()` (in other words, if your platform is UNIX) you can use the list form.

See *Safe Pipe Opens in [perlipc](#)* for more examples of this.

Beginning with v5.6.0, Perl will attempt to flush all files opened for output before any operation that may do a fork, but this may not be supported on some platforms (see *[perlport](#)*). To be safe, you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush()` method of `IO::Handle` on any open handles.

On systems that support a `close-on-exec` flag on files, the flag will be set for the newly opened file descriptor as determined by the value of `$^F`. See *[\\$^F](#)*.

Closing any piped filehandle causes the parent process to wait for the child to finish, and returns the status value in `$?`.

The filename passed to 2-argument (or 1-argument) form of `open()` will have leading and trailing whitespace deleted, and the normal redirection characters honored. This property, known as "magic open", can often be used to good effect. A user could specify a filename of *"rsh cat file |"*, or you could change certain filenames as needed:

```
$filename =~ s/(.*\.gz)\s*/$1/gzip -dc < $1|/;
open(FH, $filename) or die "Can't open $filename: $!";
```

Use 3-argument form to open a file with arbitrary weird characters in it,

```
open(FOO, '<', $file);
```

otherwise it's necessary to protect any leading and trailing whitespace:

```
$file =~ s#^\s#\s#/$1#;
open(FOO, "< $file\0");
```

(this may not work on some bizarre filesystems). One should conscientiously choose between the *magic* and 3-arguments form of `open()`:

```
open IN, $ARGV[0];
```

will allow the user to specify an argument of the form *"rsh cat file |"*, but will not work on a filename which happens to have a trailing space, while

```
open IN, '<', $ARGV[0];
```

will have exactly the opposite restrictions.

If you want a "real" C `open` (see *[open\(2\)](#)* on your system), then you should use the `sysopen` function, which involves no such magic (but may use subtly different filemodes than Perl `open()`, which is mapped to C `fopen()`). This is another way to protect your filenames from interpretation. For example:

```
use IO::Handle;
sysopen(HANDLE, $path, O_RDWR|O_CREAT|O_EXCL)
    or die "sysopen $path: $!";
$oldfh = select(HANDLE); $| = 1; select($oldfh);
print HANDLE "stuff $$\n";
seek(HANDLE, 0, 0);
print "File contains: ", <HANDLE>;
```

Using the constructor from the `IO::Handle` package (or one of its subclasses, such as `IO::File` or `IO::Socket`), you can generate anonymous filehandles that have the scope of whatever variables hold references to them, and automatically close whenever and however you

leave that scope:

```
use IO::File;
#...
sub read_myfile_munged {
    my $ALL = shift;
    my $handle = new IO::File;
    open($handle, "myfile") or die "myfile: $!";
    $first = <$handle>
        or return ();      # Automatically closed here.
    mung $first or die "mung failed";      # Or here.
    return $first, <$handle> if $ALL;      # Or here.
    $first;                 # Or here.
}
```

See [/seek](#) for some details about mixing reading and writing.

opendir DIRHANDLE,EXPR

Opens a directory named EXPR for processing by `readdir`, `telldir`, `seekdir`, `rewinddir`, and `closedir`. Returns true if successful. DIRHANDLES have their own namespace separate from FILEHANDLES.

ord EXPR

Returns the numeric (the native 8-bit encoding, like ASCII or EBCDIC, or Unicode) value of the first character of EXPR. If EXPR is omitted, uses `$_`.

For the reverse, see [/chr](#). See [perlunicode](#) and [encoding](#) for more about Unicode.

our EXPR

our EXPR TYPE

our EXPR : ATTRS

our TYPE EXPR : ATTRS

An `our` declares the listed variables to be valid globals within the enclosing block, file, or `eval`. That is, it has the same scoping rules as a "my" declaration, but does not create a local variable. If more than one value is listed, the list must be placed in parentheses. The `our` declaration has no semantic effect unless "use strict vars" is in effect, in which case it lets you use the declared global variable without qualifying it with a package name. (But only within the lexical scope of the `our` declaration. In this it differs from "use vars", which is package scoped.)

An `our` declaration declares a global variable that will be visible across its entire lexical scope, even across package boundaries. The package in which the variable is entered is determined at the point of the declaration, not at the point of use. This means the following behavior holds:

```
package Foo;
our $bar;          # declares $Foo::bar for rest of lexical scope
$bar = 20;

package Bar;
print $bar;        # prints 20
```

Multiple `our` declarations in the same lexical scope are allowed if they are in different packages. If they happened to be in the same package, Perl will emit warnings if you have asked for them.

```
use warnings;
package Foo;
our $bar;          # declares $Foo::bar for rest of lexical scope
$bar = 20;

package Bar;
```

```

our $bar = 30; declares $Bar::bar for rest of lexical scope
print $bar;           # prints 30

our $bar;             # emits warning

```

An `our` declaration may also have a list of attributes associated with it.

The exact semantics and interface of `TYPE` and `ATTRS` are still evolving. `TYPE` is currently bound to the use of `fields` pragma, and attributes are handled using the `attributes` pragma, or starting from Perl 5.8.0 also via the `Attribute::Handlers` module. See ["Private Variables via my\(\)"](#) for details, and [fields](#), [attributes](#), and [Attribute::Handlers](#).

The only currently recognized `our()` attribute is `unique` which indicates that a single copy of the global is to be used by all interpreters should the program happen to be running in a multi-interpreter environment. (The default behaviour would be for each interpreter to have its own copy of the global.) Examples:

```

our @EXPORT : unique = qw(foo);
our %EXPORT_TAGS : unique = (bar => [qw(aa bb cc)]);
our $VERSION : unique = "1.00";

```

Note that this attribute also has the effect of making the global readonly when the first new interpreter is cloned (for example, when the first new thread is created).

Multi-interpreter environments can come to being either through the `fork()` emulation on Windows platforms, or by embedding perl in a multi-threaded application. The `unique` attribute does nothing in all other environments.

pack TEMPLATE,LIST

Takes a `LIST` of values and converts it into a string using the rules given by the `TEMPLATE`. The resulting string is the concatenation of the converted values. Typically, each converted value looks like its machine-level representation. For example, on 32-bit machines a converted integer may be represented by a sequence of 4 bytes.

The `TEMPLATE` is a sequence of characters that give the order and type of values, as follows:

- a A string with arbitrary binary data, will be null padded.
- A A text (ASCII) string, will be space padded.
- Z A null terminated (ASCIZ) string, will be null padded.
- b A bit string (ascending bit order inside each byte, like `vec()`).
- B A bit string (descending bit order inside each byte).
- h A hex string (low nybble first).
- H A hex string (high nybble first).
- c A signed char value.
- C An unsigned char value. Only does bytes. See `U` for Unicode.
- s A signed short value.
- S An unsigned short value.
(This 'short' is `_exactly_` 16 bits, which may differ from what a local C compiler calls 'short'. If you want native-length shorts, use the `!` suffix.)
- i A signed integer value.
- I An unsigned integer value.
(This 'integer' is `_at_least_` 32 bits wide. Its exact size depends on what a local C compiler calls 'int', and may even be larger than the 'long' described in the next item.)

- l A signed long value.
- L An unsigned long value.
(This 'long' is `_exactly_` 32 bits, which may differ from what a local C compiler calls 'long'. If you want native-length longs, use the '!' suffix.)
- n An unsigned short in "network" (big-endian) order.
- N An unsigned long in "network" (big-endian) order.
- v An unsigned short in "VAX" (little-endian) order.
- V An unsigned long in "VAX" (little-endian) order.
(These 'shorts' and 'longs' are `_exactly_` 16 bits and `_exactly_` 32 bits, respectively.)
- q A signed quad (64-bit) value.
- Q An unsigned quad value.
(Quads are available only if your system supports 64-bit integer values `_and_` if Perl has been compiled to support those. Causes a fatal error otherwise.)
- j A signed integer value (a Perl internal integer, IV).
- J An unsigned integer value (a Perl internal unsigned integer, UV).
- f A single-precision float in the native format.
- d A double-precision float in the native format.
- F A floating point value in the native native format
(a Perl internal floating point value, NV).
- D A long double-precision float in the native format.
(Long doubles are available only if your system supports long double values `_and_` if Perl has been compiled to support those. Causes a fatal error otherwise.)
- p A pointer to a null-terminated string.
- P A pointer to a structure (fixed-length string).
- u A uuencoded string.
- U A Unicode character number. Encodes to UTF-8 internally
(or UTF-EBCDIC in EBCDIC platforms).
- w A BER compressed integer. Its bytes represent an unsigned integer in base 128, most significant digit first, with as few digits as possible. Bit eight (the high bit) is set on each byte except the last.
- x A null byte.
- X Back up a byte.
- @ Null fill to absolute position.
- (Start of a ()-group.

The following rules apply:

- Each letter may optionally be followed by a number giving a repeat count. With all types except a, A, Z, b, B, h, H, @, x, X and P the pack function will gobble up that many values from the LIST. A * for the repeat count means to use however many items are left, except for @, x, X, where it is equivalent to , and u, where it is equivalent to 1 (or 45, what is the same). A numeric repeat count may optionally be enclosed in brackets, as in `pack 'C[80]', @arr`.

One can replace the numeric repeat count by a template enclosed in brackets; then the packed length of this template in bytes is used as a count. For example, `x[L]`

skips a long (it skips the number of bytes in a long); the template `$t x[$t] $t` `unpack()`s twice what `$t` unpacks. If the template in brackets contains alignment commands (such as `x![d]`), its packed length is calculated as if the start of the template has the maximal possible alignment.

When used with `Z`, `*` results in the addition of a trailing null byte (so the packed result will be one longer than the byte length of the item).

The repeat count for `u` is interpreted as the maximal number of bytes to encode per line of output, with 0 and 1 replaced by 45.

- The `a`, `A`, and `Z` types gobble just one value, but pack it as a string of length count, padding with nulls or spaces as necessary. When unpacking, `A` strips trailing spaces and nulls, `Z` strips everything after the first null, and `a` returns data verbatim. When packing, `a`, and `Z` are equivalent.

If the value-to-pack is too long, it is truncated. If too long and an explicit count is provided, `Z` packs only `$count-1` bytes, followed by a null byte. Thus `Z` always packs a trailing null byte under all circumstances.

- Likewise, the `b` and `B` fields pack a string that many bits long. Each byte of the input field of `pack()` generates 1 bit of the result. Each result bit is based on the least-significant bit of the corresponding input byte, i.e., on `ord($byte)%2`. In particular, bytes `"0"` and `"1"` generate bits 0 and 1, as do bytes `"\0"` and `"\1"`.

Starting from the beginning of the input string of `pack()`, each 8-tuple of bytes is converted to 1 byte of output. With format `b` the first byte of the 8-tuple determines the least-significant bit of a byte, and with format `B` it determines the most-significant bit of a byte.

If the length of the input string is not exactly divisible by 8, the remainder is packed as if the input string were padded by null bytes at the end. Similarly, during `unpack()`ing the "extra" bits are ignored.

If the input string of `pack()` is longer than needed, extra bytes are ignored. A `*` for the repeat count of `pack()` means to use all the bytes of the input field. On `unpack()`ing the bits are converted to a string of `"0"`s and `"1"`s.

- The `h` and `H` fields pack a string that many nybbles (4-bit groups, representable as hexadecimal digits, 0–9a–f) long.

Each byte of the input field of `pack()` generates 4 bits of the result. For non-alphabetical bytes the result is based on the 4 least-significant bits of the input byte, i.e., on `ord($byte)%16`. In particular, bytes `"0"` and `"1"` generate nybbles 0 and 1, as do bytes `"\0"` and `"\1"`. For bytes `"a".."f"` and `"A".."F"` the result is compatible with the usual hexadecimal digits, so that `"a"` and `"A"` both generate the nybble `0xa==10`. The result for bytes `"g".."z"` and `"G".."Z"` is not well-defined.

Starting from the beginning of the input string of `pack()`, each pair of bytes is converted to 1 byte of output. With format `h` the first byte of the pair determines the least-significant nybble of the output byte, and with format `H` it determines the most-significant nybble.

If the length of the input string is not even, it behaves as if padded by a null byte at the end. Similarly, during `unpack()`ing the "extra" nybbles are ignored.

If the input string of `pack()` is longer than needed, extra bytes are ignored. A `*` for the repeat count of `pack()` means to use all the bytes of the input field. On `unpack()`ing the bits are converted to a string of hexadecimal digits.

- The `p` type packs a pointer to a null-terminated string. You are responsible for ensuring the string is not a temporary value (which can potentially get deallocated before you get around to using the packed result). The `P` type packs a pointer to a structure of the size indicated by the length. A `NULL` pointer is created if the corresponding value for `p` or `P` is `undef`, similarly for `unpack()`.
- The `/` template character allows packing and unpacking of strings where the packed structure contains a byte count followed by the string itself. You write *length-item/string-item*.

The *length-item* can be any pack template letter, and describes how the length value is packed. The ones likely to be of most use are integer-packing ones like `n` (for Java strings), `w` (for ASN.1 or SNMP) and `N` (for Sun XDR).

The *string-item* must, at present, be `"A*"`, `"a*"` or `"Z*"`. For `unpack` the length of the string is obtained from the *length-item*, but if you put in the `*` it will be ignored.

```
unpack 'C/a', "\04Gurusamy";           gives 'Guru'
unpack 'a3/A* A*', '007 Bond J ';      gives (' Bond', 'J')
pack 'n/a* w/a*', 'hello, ', 'world';  gives "\000\006hello,\005worl
```

The *length-item* is not returned explicitly from `unpack`.

Adding a count to the *length-item* letter is unlikely to do anything useful, unless that letter is `A`, `a` or `Z`. Packing with a *length-item* of `a` or `Z` may introduce `"\000"` characters, which Perl does not regard as legal in numeric strings.

- The integer types `s`, `S`, `l`, and `L` may be immediately followed by a `!` suffix to signify native shorts or longs—as you can see from above for example a bare `l` does mean exactly 32 bits, the native long (as seen by the local `C` compiler) may be larger. This is an issue mainly in 64-bit platforms. You can see whether using `!` makes any difference by

```
print length(pack("s")), " ", length(pack("s!")), "\n";
print length(pack("l")), " ", length(pack("l!")), "\n";
```

`i!` and `I!` also work but only because of completeness; they are identical to `i` and `I`.

The actual sizes (in bytes) of native shorts, ints, longs, and long longs on the platform where Perl was built are also available via [Config](#):

```
use Config;
print $Config{shortsize}, "\n";
print $Config{intsize}, "\n";
print $Config{longsize}, "\n";
print $Config{longlongsize}, "\n";
```

(The `$Config{longlongsize}` will be `undef` if your system does not support long longs.)

- The integer formats `s`, `S`, `i`, `I`, `l`, `L`, `j`, and `J` are inherently non-portable between processors and operating systems because they obey the native byteorder and endianness. For example a 4-byte integer `0x12345678` (305419896 decimal) would be ordered natively (arranged in and handled by the CPU registers) into bytes as

```
0x12 0x34 0x56 0x78      # big-endian
0x78 0x56 0x34 0x12      # little-endian
```

Basically, the Intel and VAX CPUs are little-endian, while everybody else, for

example Motorola m68k/88k, PPC, Sparc, HP PA, Power, and Cray are big-endian. Alpha and MIPS can be either: Digital/Compaq used/uses them in little-endian mode; SGI/Cray uses them in big-endian mode.

The names ‘big-endian’ and ‘little-endian’ are comic references to the classic "Gulliver's Travels" (via the paper "On Holy Wars and a Plea for Peace" by Danny Cohen, USC/ISI IEN 137, April 1, 1980) and the egg-eating habits of the Lilliputians.

Some systems may have even weirder byte orders such as

```
0x56 0x78 0x12 0x34
0x34 0x12 0x78 0x56
```

You can see your system's preference with

```
print join(" ", map { sprintf "%#02x", $_ }
             unpack("C*",pack("L",0x12345678))), "\n";
```

The byteorder on the platform where Perl was built is also available via [Config](#):

```
use Config;
print $Config{byteorder}, "\n";
```

Byteorders ‘1234’ and ‘12345678’ are little-endian, ‘4321’ and ‘87654321’ are big-endian.

If you want portable packed integers use the formats n, N, v, and V, their byte endianness and size are known. See also [perlport](#).

- Real numbers (floats and doubles) are in the native machine format only; due to the multiplicity of floating formats around, and the lack of a standard "network" representation, no facility for interchange has been made. This means that packed floating point data written on one machine may not be readable on another – even if both use IEEE floating point arithmetic (as the endian-ness of the memory representation is not part of the IEEE spec). See also [perlport](#).

Note that Perl uses doubles internally for all numeric calculation, and converting from double into float and thence back to double again will lose precision (i.e., `unpack("f", pack("f", $foo))` will not in general equal `$foo`).

- If the pattern begins with a U, the resulting string will be treated as Unicode-encoded. You can force UTF8 encoding on in a string with an initial U0, and the bytes that follow will be interpreted as Unicode characters. If you don't want this to happen, you can begin your pattern with C0 (or anything else) to force Perl not to UTF8 encode your string, and then follow this with a U* somewhere in your pattern.
- You must yourself do any alignment or padding by inserting for example enough 'x'es while packing. There is no way to `pack()` and `unpack()` could know where the bytes are going to or coming from. Therefore `pack` (and `unpack`) handle their output and input as flat sequences of bytes.
- A `()`-group is a sub-TEMPLATE enclosed in parentheses. A group may take a repeat count, both as postfix, and via the `/` template character.
- `x` and `X` accept `!` modifier. In this case they act as alignment commands: they jump forward/back to the closest position aligned at a multiple of count bytes. For example, to `pack()` or `unpack()` C's `struct {char c; double d; char cc[2]}` one may need to use the template `C x![d] d C[2]`; this assumes that doubles must be aligned on the double's size.

For alignment commands count of 0 is equivalent to count of 1; both result in no-ops.

- A comment in a TEMPLATE starts with # and goes to the end of line.
- If TEMPLATE requires more arguments to pack() than actually given, pack() assumes additional "" arguments. If TEMPLATE requires less arguments to pack() than actually given, extra arguments are ignored.

Examples:

```
$foo = pack("CCCC",65,66,67,68);
# foo eq "ABCD"
$foo = pack("C4",65,66,67,68);
# same thing
$foo = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
# same thing with Unicode circled letters

$foo = pack("ccxxcc",65,66,67,68);
# foo eq "AB\0\0CD"

# note: the above examples featuring "C" and "c" are true
# only on ASCII and ASCII-derived systems such as ISO Latin 1
# and UTF-8. In EBCDIC the first example would be
# $foo = pack("CCCC",193,194,195,196);

$foo = pack("s2",1,2);
# "\1\0\2\0" on little-endian
# "\0\1\0\2" on big-endian

$foo = pack("a4","abcd","x","y","z");
# "abcd"

$foo = pack("aaaa","abcd","x","y","z");
# "axyz"

$foo = pack("a14","abcdefg");
# "abcdefg\0\0\0\0\0\0\0\0"

$foo = pack("i9pl", gmtime);
# a real struct tm (on my system anyway)

$utmp_template = "Z8 Z8 Z16 L";
$utmp = pack($utmp_template, @utmp1);
# a struct utmp (BSDish)

@utmp2 = unpack($utmp_template, $utmp);
# "@utmp1" eq "@utmp2"

sub bintodec {
    unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}

$foo = pack('sx2l', 12, 34);
# short 12, two zero bytes padding, long 34
$bar = pack('s@4l', 12, 34);
# short 12, zero fill to position 4, long 34
# $foo eq $bar
```

The same template may generally also be used in unpack().

package NAMESPACE

package Declares the compilation unit as being in the given namespace. The scope of the package declaration is from the declaration itself through the end of the enclosing block, file, or eval (the same as the `my` operator). All further unqualified dynamic identifiers will be in this namespace. A package statement affects only dynamic variables—including those you’ve used `local` on—but *not* lexical variables, which are created with `my`. Typically it would be the first declaration in a file to be included by the `require` or `use` operator. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: `$Package::Variable`. If the package name is null, the main package is assumed. That is, `$::sail` is equivalent to `$main::sail` (as well as to `$main`sail`, still seen in older code).

If `NAMESPACE` is omitted, then there is no current package, and all identifiers must be fully qualified or lexicals. However, you are strongly advised not to make use of this feature. Its use can cause unexpected behaviour, even crashing some versions of Perl. It is deprecated, and will be removed from a future release.

See [Packages in perlmod](#) for more information about packages, modules, and classes. See [perlsub](#) for other scoping issues.

pipe READHANDLE,WRITEHANDLE

Opens a pair of connected pipes like the corresponding system call. Note that if you set up a loop of piped processes, deadlock can occur unless you are very careful. In addition, note that Perl’s pipes use IO buffering, so you may need to set `$|` to flush your `WRITEHANDLE` after each command, depending on the application.

See [IPC::Open2](#), [IPC::Open3](#), and [Bidirectional Communication in perlipc](#) for examples of such things.

On systems that support a `close-on-exec` flag on files, the flag will be set for the newly opened file descriptors as determined by the value of `$^F`. See [\\$^F](#).

pop ARRAY

pop Pops and returns the last value of the array, shortening the array by one element. Has an effect similar to

```
$ARRAY[$#ARRAY--]
```

If there are no elements in the array, returns the undefined value (although this may happen at other times as well). If `ARRAY` is omitted, pops the `@ARGV` array in the main program, and the `@_` array in subroutines, just like `shift`.

pos SCALAR

pos Returns the offset of where the last `m//g` search left off for the variable in question (`$_` is used when the variable is not specified). May be modified to change that offset. Such modification will also influence the `\G` zero-width assertion in regular expressions. See [perlre](#) and [perlop](#).

print FILEHANDLE LIST**print LIST**

print Prints a string or a list of strings. Returns true if successful. `FILEHANDLE` may be a scalar variable name, in which case the variable contains the name of or a reference to the filehandle, thus introducing one level of indirection. (NOTE: If `FILEHANDLE` is a variable and the next token is a term, it may be misinterpreted as an operator unless you interpose a `+` or put parentheses around the arguments.) If `FILEHANDLE` is omitted, prints by default to standard output (or to the last selected output channel—see [/select](#)). If `LIST` is also omitted, prints `$_` to the currently selected output channel. To set the default output channel to something other than `STDOUT` use the `select` operation. The current value of `$,` (if any) is printed between each `LIST` item. The current value of `$\` (if any) is printed after the entire `LIST` has been printed.

Because `print` takes a `LIST`, anything in the `LIST` is evaluated in list context, and any subroutine that you call will have one or more of its expressions evaluated in list context. Also be careful not to follow the `print` keyword with a left parenthesis unless you want the corresponding right parenthesis to terminate the arguments to the `print`—interpose a `+` or put parentheses around all the arguments.

Note that if you're storing `FILEHANDLES` in an array or other expression, you will have to use a block returning its value instead:

```
print { $files[$i] } "stuff\n";
print { $OK ? STDOUT : STDERR } "stuff\n";
```

`printf FILEHANDLE FORMAT, LIST`

`printf FORMAT, LIST`

Equivalent to `print FILEHANDLE sprintf(FORMAT, LIST)`, except that `$\` (the output record separator) is not appended. The first argument of the list will be interpreted as the `printf` format. See `sprintf` for an explanation of the format argument. If `use locale` is in effect, the character used for the decimal point in formatted real numbers is affected by the `LC_NUMERIC` locale. See [perllocale](#).

Don't fall into the trap of using a `printf` when a simple `print` would do. The `print` is more efficient and less error prone.

`prototype FUNCTION`

Returns the prototype of a function as a string (or `undef` if the function has no prototype). `FUNCTION` is a reference to, or the name of, the function whose prototype you want to retrieve.

If `FUNCTION` is a string starting with `CORE::`, the rest is taken as a name for Perl builtin. If the builtin is not *overridable* (such as `qw//`) or its arguments cannot be expressed by a prototype (such as `system`) returns `undef` because the builtin does not really behave like a Perl function. Otherwise, the string describing the equivalent prototype is returned.

`push ARRAY,LIST`

Treats `ARRAY` as a stack, and pushes the values of `LIST` onto the end of `ARRAY`. The length of `ARRAY` increases by the length of `LIST`. Has the same effect as

```
for $value (LIST) {
    $ARRAY[++$#ARRAY] = $value;
}
```

but is more efficient. Returns the new number of elements in the array.

`q/STRING/`

`qq/STRING/`

`qr/STRING/`

`qx/STRING/`

`qw/STRING/`

Generalized quotes. See [Regex Quote-Like Operators in perlop](#).

`quotemeta EXPR`

`quotemeta`

Returns the value of `EXPR` with all non-"word" characters backslashed. (That is, all characters not matching `/[A-Za-z_0-9]/` will be preceded by a backslash in the returned string, regardless of any locale settings.) This is the internal function implementing the `\Q` escape in double-quoted strings.

If `EXPR` is omitted, uses `$_`.

rand EXPR

rand Returns a random fractional number greater than or equal to 0 and less than the value of EXPR. (EXPR should be positive.) If EXPR is omitted, the value 1 is used. Currently EXPR with the value 0 is also special-cased as 1 – this has not been documented before perl 5.8.0 and is subject to change in future versions of perl. Automatically calls `srand` unless `srand` has already been called. See also `srand`.

Apply `int()` to the value returned by `rand()` if you want random integers instead of random fractional numbers. For example,

```
int(rand(10))
```

returns a random integer between 0 and 9, inclusive.

(Note: If your `rand` function consistently returns numbers that are too large or too small, then your version of Perl was probably compiled with the wrong number of `RANDBITS`.)

read FILEHANDLE, SCALAR, LENGTH, OFFSET**read** FILEHANDLE, SCALAR, LENGTH

Attempts to read *LENGTH* *characters* of data into variable *SCALAR* from the specified *FILEHANDLE*. Returns the number of characters actually read, at end of file, or `undef` if there was an error. *SCALAR* will be grown or shrunk to the length actually read. If *SCALAR* needs growing, the new bytes will be zero bytes. An *OFFSET* may be specified to place the read data into some other place in *SCALAR* than the beginning. The call is actually implemented in terms of either Perl's or system's `fread()` call. To get a true `read(2)` system call, see `sysread`.

Note the *characters*: depending on the status of the filehandle, either (8-bit) bytes or characters are read. By default all filehandles operate on bytes, but for example if the filehandle has been opened with the `:utf8` I/O layer (see [/open](#), and the `open` pragma, [open](#)), the I/O will operate on characters, not bytes.

readdir DIRHANDLE

Returns the next directory entry for a directory opened by `opendir`. If used in list context, returns all the rest of the entries in the directory. If there are no more entries, returns an undefined value in scalar context or a null list in list context.

If you're planning to filetest the return values out of a `readdir`, you'd better prepend the directory in question. Otherwise, because we didn't `chdir` there, it would have been testing the wrong file.

```
opendir(DIR, $some_dir) || die "can't opendir $some_dir: $!";
@dots = grep { /^\.\/ && -f "$some_dir/$_" } readdir(DIR);
closedir DIR;
```

readline EXPR

Reads from the filehandle whose typeglob is contained in EXPR. In scalar context, each call reads and returns the next line, until end-of-file is reached, whereupon the subsequent call returns `undef`. In list context, reads until end-of-file is reached and returns a list of lines. Note that the notion of "line" used here is however you may have defined it with `$/` or `$INPUT_RECORD_SEPARATOR`. See [\\$/](#) in [perlvar](#).

When `$/` is set to `undef`, when `readline()` is in scalar context (i.e. file slurp mode), and when an empty file is read, it returns `''` the first time, followed by `undef` subsequently.

This is the internal function implementing the `< <EXPR` operator, but you can use it directly. The `< <EXPR` operator is discussed in more detail in [I/O Operators in perlop](#).

```
$line = <STDIN>;
$line = readline(*STDIN);           # same thing
```

readlink *EXPR*

readlink Returns the value of a symbolic link, if symbolic links are implemented. If not, gives a fatal error. If there is some system error, returns the undefined value and sets \$! (errno). If *EXPR* is omitted, uses \$_.

readpipe *EXPR*

EXPR is executed as a system command. The collected standard output of the command is returned. In scalar context, it comes back as a single (potentially multi-line) string. In list context, returns a list of lines (however you've defined lines with \$/ or \$INPUT_RECORD_SEPARATOR). This is the internal function implementing the qx/*EXPR*/ operator, but you can use it directly. The qx/*EXPR*/ operator is discussed in more detail in *I/O Operators in perlop*.

recv *SOCKET, SCALAR, LENGTH, FLAGS*

Receives a message on a socket. Attempts to receive *LENGTH* characters of data into variable *SCALAR* from the specified *SOCKET* filehandle. *SCALAR* will be grown or shrunk to the length actually read. Takes the same flags as the system call of the same name. Returns the address of the sender if *SOCKET*'s protocol supports this; returns an empty string otherwise. If there's an error, returns the undefined value. This call is actually implemented in terms of *recvfrom*(2) system call. See *UDP: Message Passing in perlipc* for examples.

Note the *characters*: depending on the status of the socket, either (8-bit) bytes or characters are received. By default all sockets operate on bytes, but for example if the socket has been changed using *binmode*() to operate with the :utf8 I/O layer (see the *open* pragma, *open*), the I/O will operate on characters, not bytes.

redo *LABEL*

redo The *redo* command restarts the loop block without evaluating the conditional again. The *continue* block, if any, is not executed. If the *LABEL* is omitted, the command refers to the innermost enclosing loop. This command is normally used by programs that want to lie to themselves about what was just input:

```
# a simpleminded Pascal comment stripper
# (warning: assumes no { or } in strings)
LINE: while (<STDIN>) {
    while (s|({.*}.*){.*}|$1 |) {}
    s|{.*}| |;
    if (s|{.*}| |) {
        $front = $_;
        while (<STDIN>) {
            if (/}/) {          # end of comment?
                s|^|$front\| |;
                redo LINE;
            }
        }
    }
}
print;
```

redo cannot be used to retry a block which returns a value such as *eval* {}, *sub* {} or *do* {}, and should not be used to exit a *grep*() or *map*() operation.

Note that a block by itself is semantically identical to a loop that executes once. Thus *redo* inside such a block will effectively turn it into a looping construct.

See also */continue* for an illustration of how *last*, *next*, and *redo* work.

ref **EXPR**

ref Returns a true value if EXPR is a reference, false otherwise. If EXPR is not specified, `$_` will be used. The value returned depends on the type of thing the reference is a reference to. Builtin types include:

```
SCALAR
ARRAY
HASH
CODE
REF
GLOB
LVALUE
```

If the referenced object has been blessed into a package, then that package name is returned instead. You can think of `ref` as a `typeof` operator.

```
if (ref($r) eq "HASH") {
    print "r is a reference to a hash.\n";
}
unless (ref($r)) {
    print "r is not a reference at all.\n";
}
if (UNIVERSAL::isa($r, "HASH")) { # for subclassing
    print "r is a reference to something that isa hash.\n";
}
```

See also [perlref](#).

rename **OLDNAME,NEWNAME**

Changes the name of a file; an existing file `NEWNAME` will be clobbered. Returns true for success, false otherwise.

Behavior of this function varies wildly depending on your system implementation. For example, it will usually not work across file system boundaries, even though the system `mv` command sometimes compensates for this. Other restrictions include whether it works on directories, open files, or pre-existing files. Check [perlport](#) and either the `rename(2)` manpage or equivalent system documentation for details.

require **VERSION****require** **EXPR**

require Demands a version of Perl specified by `VERSION`, or demands some semantics specified by `EXPR` or by `$_` if `EXPR` is not supplied.

`VERSION` may be either a numeric argument such as 5.006, which will be compared to `$]`, or a literal of the form `v5.6.1`, which will be compared to `$^V` (aka `$PERL_VERSION`). A fatal error is produced at run time if `VERSION` is greater than the version of the current Perl interpreter. Compare with [/use](#), which can do a similar check at compile time.

Specifying `VERSION` as a literal of the form `v5.6.1` should generally be avoided, because it leads to misleading error messages under earlier versions of Perl which do not support this syntax. The equivalent numeric version should be used instead.

```
require v5.6.1;      # run time version check
require 5.6.1;       # ditto
require 5.006_001;   # ditto; preferred for backwards compatibility
```

Otherwise, demands that a library file be included if it hasn't already been included. The file is included via the `do-FILE` mechanism, which is essentially just a variety of `eval`. Has semantics similar to the following subroutine:

```

sub require {
    my($filename) = @_;
    return 1 if $INC{$filename};
    my($realfilename,$result);
    ITER: {
        foreach $prefix (@INC) {
            $realfilename = "$prefix/$filename";
            if (-f $realfilename) {
                $INC{$filename} = $realfilename;
                $result = do $realfilename;
                last ITER;
            }
        }
        die "Can't find $filename in \@INC";
    }
    delete $INC{$filename} if $@ || !$result;
    die $@ if $@;
    die "$filename did not return true value" unless $result;
    return $result;
}

```

Note that the file will not be included twice under the same specified name. The file must return true as the last statement to indicate successful execution of any initialization code, so it's customary to end such a file with `1;` unless you're sure it'll return true otherwise. But it's better just to put the `1;`, in case you add more statements.

If `EXPR` is a bareword, the `require` assumes a ***.pm*** extension and replaces `::` with `/` in the filename for you, to make it easy to load standard modules. This form of loading of modules does not risk altering your namespace.

In other words, if you try this:

```
require Foo::Bar;    # a splendid bareword
```

The `require` function will actually look for the ***Foo/Bar.pm*** file in the directories specified in the `@INC` array.

But if you try this:

```

$class = 'Foo::Bar';
require $class;    # $class is not a bareword
#or
require "Foo::Bar"; # not a bareword because of the "

```

The `require` function will look for the ***Foo::Bar*** file in the `@INC` array and will complain about not finding ***Foo::Bar*** there. In this case you can do:

```
eval "require $class";
```

You can also insert hooks into the import facility, by putting directly Perl code into the `@INC` array. There are three forms of hooks: subroutine references, array references and blessed objects.

Subroutine references are the simplest case. When the inclusion system walks through `@INC` and encounters a subroutine, this subroutine gets called with two parameters, the first being a reference to itself, and the second the name of the file to be included (e.g. ***Foo/Bar.pm***). The subroutine should return `undef` or a filehandle, from which the file to include will be read. If `undef` is returned, `require` will look at the remaining elements of `@INC`.

If the hook is an array reference, its first element must be a subroutine reference. This

subroutine is called as above, but the first parameter is the array reference. This enables to pass indirectly some arguments to the subroutine.

In other words, you can write:

```
push @INC, \&my_sub;
sub my_sub {
    my ($coderef, $filename) = @_; # $coderef is \&my_sub
    ...
}
```

or:

```
push @INC, [ \&my_sub, $x, $y, ... ];
sub my_sub {
    my ($arrayref, $filename) = @_;
    # Retrieve $x, $y, ...
    my @parameters = @$arrayref[1..$#$arrayref];
    ...
}
```

If the hook is an object, it must provide an INC method, that will be called as above, the first parameter being the object itself. (Note that you must fully qualify the sub's name, as it is always forced into package main.) Here is a typical code layout:

```
# In Foo.pm
package Foo;
sub new { ... }
sub Foo::INC {
    my ($self, $filename) = @_;
    ...
}

# In the main program
push @INC, new Foo(...);
```

Note that these hooks are also permitted to set the %INC entry corresponding to the files they have loaded. See [%INC](#).

For a yet-more-powerful import facility, see [/use](#) and [perlmod](#).

reset EXPR

reset Generally used in a `continue` block at the end of a loop to clear variables and reset ?? searches so that they work again. The expression is interpreted as a list of single characters (hyphens allowed for ranges). All variables and arrays beginning with one of those letters are reset to their pristine state. If the expression is omitted, one-match searches (?pattern?) are reset to match again. Resets only variables or searches in the current package. Always returns 1. Examples:

```
reset 'X';           # reset all X variables
reset 'a-z';         # reset lower case variables
reset;               # just reset ?one-time? searches
```

Resetting "A-Z" is not recommended because you'll wipe out your @ARGV and @INC arrays and your %ENV hash. Resets only package variables—lexical variables are unaffected, but they clean themselves up on scope exit anyway, so you'll probably want to use them instead. See [/my](#).

return EXPR

return Returns from a subroutine, `eval`, or `do FILE` with the value given in `EXPR`. Evaluation of `EXPR` may be in list, scalar, or void context, depending on how the return value will be used, and the context may vary from one execution to the next (see `wantarray`). If no `EXPR` is given, returns an empty list in list context, the undefined value in scalar context, and (of course) nothing at all in a void context.

(Note that in the absence of an explicit `return`, a subroutine, `eval`, or `do FILE` will automatically return the value of the last expression evaluated.)

reverse LIST

In list context, returns a list value consisting of the elements of `LIST` in the opposite order. In scalar context, concatenates the elements of `LIST` and returns a string value with all characters in the opposite order.

```
print reverse <>;           # line tac, last line first
undef $/;                  # for efficiency of <>
print scalar reverse <>;    # character tac, last line tsrif
```

This operator is also handy for inverting a hash, although there are some caveats. If a value is duplicated in the original hash, only one of those can be represented as a key in the inverted hash. Also, this has to unwind one hash and build a whole new one, which may take some time on a large hash, such as from a DBM file.

```
%by_name = reverse %by_address;    # Invert the hash
```

rewinddir DIRHANDLE

Sets the current position to the beginning of the directory for the `readdir` routine on `DIRHANDLE`.

rindex STR,SUBSTR,POSITION

rindex STR,SUBSTR

Works just like `index()` except that it returns the position of the **LAST** occurrence of `SUBSTR` in `STR`. If `POSITION` is specified, returns the last occurrence at or before that position.

rmdir FILENAME

rmdir Deletes the directory specified by `FILENAME` if that directory is empty. If it succeeds it returns true, otherwise it returns false and sets `$!` (`errno`). If `FILENAME` is omitted, uses `$_`.

s/// The substitution operator. See [perlop](#).

scalar EXPR

Forces `EXPR` to be interpreted in scalar context and returns the value of `EXPR`.

```
@counts = ( scalar @a, scalar @b, scalar @c );
```

There is no equivalent operator to force an expression to be interpolated in list context because in practice, this is never needed. If you really wanted to do so, however, you could use the construction `@{ [(some expression)] }`, but usually a simple `(some expression)` suffices.

Because `scalar` is unary operator, if you accidentally use for `EXPR` a parenthesized list, this behaves as a scalar comma expression, evaluating all but the last element in void context and returning the final element evaluated in scalar context. This is seldom what you want.

The following single statement:

```
print uc(scalar(&foo,$bar)), $baz;
```

is the moral equivalent of these two:

```
&foo;
print (uc ($bar) , $baz) ;
```

See [perlop](#) for more details on unary operators and the comma operator.

seek FILEHANDLE,POSITION,WHENCE

Sets FILEHANDLE's position, just like the `fseek` call of `stdio`. FILEHANDLE may be an expression whose value gives the name of the filehandle. The values for WHENCE are to set the new position *in bytes* to POSITION, 1 to set it to the current position plus POSITION, and 2 to set it to EOF plus POSITION (typically negative). For WHENCE you may use the constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` (start of the file, current position, end of the file) from the `Fcntl` module. Returns 1 upon success, otherwise.

Note the *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `:utf8` open layer), `tell()` will return byte offsets, not character offsets (because implementing that would render `seek()` and `tell()` rather slow).

If you want to position file for `sysread` or `syswrite`, don't use `seek`—buffering makes its effect on the file's system position unpredictable and non-portable. Use `sysseek` instead.

Due to the rules and rigors of ANSI C, on some systems you have to do a `seek` whenever you switch between reading and writing. Amongst other things, this may have the effect of calling `stdio's clearerr(3)`. A WHENCE of 1 (`SEEK_CUR`) is useful for not moving the file position:

```
seek (TEST, 0, 1) ;
```

This is also useful for applications emulating `tail -f`. Once you hit EOF on your read, and then sleep for a while, you might have to stick in a `seek()` to reset things. The `seek` doesn't change the current position, but it *does* clear the end-of-file condition on the handle, so that the next `< <FILE` makes Perl try again to read something. We hope.

If that doesn't work (some IO implementations are particularly cantankerous), then you may need something more like this:

```
for (;;) {
    for ($curpos = tell(FILE); $_ = <FILE>;
        $curpos = tell(FILE)) {
        # search for some stuff and put it into files
    }
    sleep($for_a_while);
    seek(FILE, $curpos, 0);
}
```

seekdir DIRHANDLE,POS

Sets the current position for the `readdir` routine on DIRHANDLE. POS must be a value returned by `telldir`. Has the same caveats about possible directory compaction as the corresponding system library routine.

select FILEHANDLE

select Returns the currently selected filehandle. Sets the current default filehandle for output, if FILEHANDLE is supplied. This has two effects: first, a `write` or a `print` without a filehandle will default to this FILEHANDLE. Second, references to variables related to output will refer to this output channel. For example, if you have to set the top of form format for more than one output channel, you might do the following:

```
select (REPORT1) ;
$^ = 'report1_top' ;
select (REPORT2) ;
$^ = 'report2_top' ;
```


FILEHANDLE may be an expression whose value gives the name of the actual filehandle. Thus:

```
$oldfh = select(STDERR); $| = 1; select($oldfh);
```

Some programmers may prefer to think of filehandles as objects with methods, preferring to write the last example as:

```
use IO::Handle;
STDERR->autoflush(1);
```

select RBITS,WBITS,EBITS,TIMEOUT

This calls the select(2) system call with the bit masks specified, which can be constructed using `fileno` and `vec`, along these lines:

```
$rin = $win = $ein = '';
vec($rin,fileno(STDIN),1) = 1;
vec($win,fileno(STDOUT),1) = 1;
$ein = $rin | $win;
```

If you want to select on many filehandles you might wish to write a subroutine:

```
sub fhbits {
    my(@fhlist) = split(' ', $_[0]);
    my($bits);
    for (@fhlist) {
        vec($bits,fileno($_),1) = 1;
    }
    $bits;
}
$rin = fhbits('STDIN TTY SOCK');
```

The usual idiom is:

```
($nfound,$timeleft) =
    select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

or to block until something becomes ready just do this

```
$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

Most systems do not bother to return anything useful in `$timeleft`, so calling `select()` in scalar context just returns `$nfound`.

Any of the bit masks can also be `undef`. The timeout, if specified, is in seconds, which may be fractional. Note: not all implementations are capable of returning the `$timeleft`. If not, they always return `$timeleft` equal to the supplied `$timeout`.

You can effect a sleep of 250 milliseconds this way:

```
select(undef, undef, undef, 0.25);
```

Note that whether `select` gets restarted after signals (say, `SIGALRM`) is implementation-dependent.

WARNING: One should not attempt to mix buffered I/O (like `read` or `<FH`) with `select`, except as permitted by POSIX, and even then only on POSIX systems. You have to use `sysread` instead.

semctl ID,SEMNUM,CMD,ARG

Calls the System V IPC function `semctl`. You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If `CMD` is `IPC_STAT` or `GETALL`, then `ARG` must be a variable which will hold the returned `semid_ds` structure or semaphore value array. Returns like `ioctl`: the undefined value for error, "0 but true" for zero, or the actual return value otherwise. The `ARG` must consist of a vector of native short integers, which may be created with `pack("s!", (0)x$nsem)`. See also [SysV IPC in perlipc](#), `IPC::SysV`, `IPC::Semaphore` documentation.

`semget KEY,NSEMS,FLAGS`

Calls the System V IPC function `semget`. Returns the semaphore id, or the undefined value if there is an error. See also [SysV IPC in perlipc](#), `IPC::SysV`, `IPC::SysV::Semaphore` documentation.

`semop KEY,OPSTRING`

Calls the System V IPC function `semop` to perform semaphore operations such as signalling and waiting. `OPSTRING` must be a packed array of `semop` structures. Each `semop` structure can be generated with `pack("s!3", $semnum, $semop, $semflag)`. The number of semaphore operations is implied by the length of `OPSTRING`. Returns true if successful, or false if there is an error. As an example, the following code waits on semaphore `$semnum` of semaphore id `$semid`:

```
$semop = pack("s!3", $semnum, -1, 0);
die "Semaphore trouble: $!\n" unless semop($semid, $semop);
```

To signal the semaphore, replace `-1` with `1`. See also [SysV IPC in perlipc](#), `IPC::SysV`, and `IPC::SysV::Semaphore` documentation.

`send SOCKET,MSG,FLAGS,TO`

`send SOCKET,MSG,FLAGS`

Sends a message on a socket. Attempts to send the scalar `MSG` to the `SOCKET` filehandle. Takes the same flags as the system call of the same name. On unconnected sockets you must specify a destination to send `TO`, in which case it does a `C sendto`. Returns the number of characters sent, or the undefined value if there is an error. The C system call `sendmsg(2)` is currently unimplemented. See [UDP: Message Passing in perlipc](#) for examples.

Note the *characters*: depending on the status of the socket, either (8-bit) bytes or characters are sent. By default all sockets operate on bytes, but for example if the socket has been changed using `binmode()` to operate with the `:utf8` I/O layer (see [/open](#), or the open pragma, [open](#)), the I/O will operate on characters, not bytes.

`setpgrp PID,PGRP`

Sets the current process group for the specified `PID`, for the current process. Will produce a fatal error if used on a machine that doesn't implement `POSIX setpgid(2)` or `BSD setpgrp(2)`. If the arguments are omitted, it defaults to `0, 0`. Note that the BSD 4.2 version of `setpgrp` does not accept any arguments, so only `setpgrp(0, 0)` is portable. See also `POSIX::setsid()`.

`setpriority WHICH,WHO,PRIORITY`

Sets the current priority for a process, a process group, or a user. (See `setpriority(2)`.) Will produce a fatal error if used on a machine that doesn't implement `setpriority(2)`.

`setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL`

Sets the socket option requested. Returns undefined if there is an error. `OPTVAL` may be specified as `undef` if you don't want to pass an argument.

`shift ARRAY`

`shift` Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. If there are no elements in the array, returns the undefined value. If `ARRAY` is omitted, shifts the `@_` array within the lexical scope of subroutines and formats, and the

@ARGV array at file scopes or within the lexical scopes established by the `eval` `''`, `BEGIN` `{}`, `INIT` `{}`, `CHECK` `{}`, and `END` `{}` constructs.

See also `unshift`, `push`, and `pop`. `shift` and `unshift` do the same thing to the left end of an array that `pop` and `push` do to the right end.

shmctl ID,CMD,ARG

Calls the System V IPC function `shmctl`. You'll probably have to say

```
use IPC::SysV;
```

first to get the correct constant definitions. If `CMD` is `IPC_STAT`, then `ARG` must be a variable which will hold the returned `shmids` structure. Returns like `ioctl`: the undefined value for error, "but true" for zero, or the actual return value otherwise. See also [SysV IPC in *perlipc*](#) and `IPC::SysV` documentation.

shmget KEY,SIZE,FLAGS

Calls the System V IPC function `shmget`. Returns the shared memory segment id, or the undefined value if there is an error. See also [SysV IPC in *perlipc*](#) and `IPC::SysV` documentation.

shmread ID,VAR,POS,SIZE

shmwrite ID,STRING,POS,SIZE

Reads or writes the System V shared memory segment ID starting at position `POS` for size `SIZE` by attaching to it, copying in/out, and detaching from it. When reading, `VAR` must be a variable that will hold the data read. When writing, if `STRING` is too long, only `SIZE` bytes are used; if `STRING` is too short, nulls are written to fill out `SIZE` bytes. Return true if successful, or false if there is an error. `shmread()` taints the variable. See also [SysV IPC in *perlipc*](#), `IPC::SysV` documentation, and the `IPC::Shareable` module from CPAN.

shutdown SOCKET,HOW

Shuts down a socket connection in the manner indicated by `HOW`, which has the same interpretation as in the system call of the same name.

```
shutdown(SOCKET, 0);    # I/we have stopped reading data
shutdown(SOCKET, 1);    # I/we have stopped writing data
shutdown(SOCKET, 2);    # I/we have stopped using this socket
```

This is useful with sockets when you want to tell the other side you're done writing but not done reading, or vice versa. It's also a more insistent form of close because it also disables the file descriptor in any forked copies in other processes.

sin EXPR

`sin` Returns the sine of `EXPR` (expressed in radians). If `EXPR` is omitted, returns sine of `$_`.

For the inverse sine operation, you may use the `Math::Trig::asin` function, or use this relation:

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

sleep EXPR

`sleep` Causes the script to sleep for `EXPR` seconds, or forever if no `EXPR`. May be interrupted if the process receives a signal such as `SIGALRM`. Returns the number of seconds actually slept. You probably cannot mix alarm and `sleep` calls, because `sleep` is often implemented using alarm.

On some older systems, it may sleep up to a full second less than what you requested, depending on how it counts seconds. Most modern systems always sleep the full amount. They may appear to sleep longer than that, however, because your process might not be scheduled right away in a busy multitasking system.

For delays of finer granularity than one second, you may use Perl's `syscall` interface to access `setitimer(2)` if your system supports it, or else see [/select](#) above. The `Time::HiRes` module (from CPAN, and starting from Perl 5.8 part of the standard distribution) may also help.

See also the `POSIX` module's `pause` function.

`socket SOCKET,DOMAIN,TYPE,PROTOCOL`

Opens a socket of the specified kind and attaches it to filehandle `SOCKET`. `DOMAIN`, `TYPE`, and `PROTOCOL` are specified the same as for the system call of the same name. You should use `Socket` first to get the proper definitions imported. See the examples in [Sockets: Client/Server Communication in *perlipc*](#).

On systems that support a `close-on-exec` flag on files, the flag will be set for the newly opened file descriptor, as determined by the value of `$^F`. See [\\$^F](#).

`socketpair SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL`

Creates an unnamed pair of sockets in the specified domain, of the specified type. `DOMAIN`, `TYPE`, and `PROTOCOL` are specified the same as for the system call of the same name. If unimplemented, yields a fatal error. Returns true if successful.

On systems that support a `close-on-exec` flag on files, the flag will be set for the newly opened file descriptors, as determined by the value of `$^F`. See [\\$^F](#).

Some systems defined `pipe` in terms of `socketpair`, in which a call to `pipe(Rdr, Wtr)` is essentially:

```
use Socket;
socketpair(Rdr, Wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown(Rdr, 1);          # no more writing for reader
shutdown(Wtr, 0);          # no more reading for writer
```

See [perlipc](#) for an example of `socketpair` use. Perl 5.8 and later will emulate `socketpair` using IP sockets to localhost if your system implements sockets but not `socketpair`.

`sort SUBNAME LIST`

`sort BLOCK LIST`

`sort LIST` In list context, this sorts the `LIST` and returns the sorted list value. In scalar context, the behaviour of `sort()` is undefined.

If `SUBNAME` or `BLOCK` is omitted, `sorts` in standard string comparison order. If `SUBNAME` is specified, it gives the name of a subroutine that returns an integer less than, equal to, or greater than , depending on how the elements of the list are to be ordered. (The `<` `<=` and `cmp` operators are extremely useful in such routines.) `SUBNAME` may be a scalar variable name (unsubscripted), in which case the value provides the name of (or a reference to) the actual subroutine to use. In place of a `SUBNAME`, you can provide a `BLOCK` as an anonymous, in-line sort subroutine.

If the subroutine's prototype is `($$)`, the elements to be compared are passed by reference in `@_`, as for a normal subroutine. This is slower than unprototyped subroutines, where the elements to be compared are passed into the subroutine as the package global variables `$a` and `$b` (see example below). Note that in the latter case, it is usually counter-productive to declare `$a` and `$b` as lexicals.

In either case, the subroutine may not be recursive. The values to be compared are always passed by reference, so don't modify them.

You also cannot exit out of the sort block or subroutine using any of the loop control operators described in [perlsyn](#) or with `goto`.

When `use locale` is in effect, `sort LIST` sorts `LIST` according to the current collation locale. See [perllocale](#).

Perl 5.6 and earlier used a quicksort algorithm to implement sort. That algorithm was not stable, and *could* go quadratic. (A *stable* sort preserves the input order of elements that compare equal. Although quicksort's run time is $O(N \log N)$ when averaged over all arrays of length N , the time can be $O(N^2)$, *quadratic* behavior, for some inputs.) In 5.7, the quicksort implementation was replaced with a stable mergesort algorithm whose worst case behavior is $O(N \log N)$. But benchmarks indicated that for some inputs, on some platforms, the original quicksort was faster. 5.8 has a sort pragma for limited control of the sort. Its rather blunt control of the underlying algorithm may not persist into future perls, but the ability to characterize the input or output in implementation independent ways quite probably will. See [/use](#).

Examples:

```
# sort lexically
@articles = sort @files;

# same thing, but with explicit sort routine
@articles = sort {$a cmp $b} @files;

# now case-insensitively
@articles = sort {uc($a) cmp uc($b)} @files;

# same thing in reversed order
@articles = sort {$b cmp $a} @files;

# sort numerically ascending
@articles = sort {$a <=> $b} @files;

# sort numerically descending
@articles = sort {$b <=> $a} @files;

# this sorts the %age hash by value instead of key
# using an in-line function
@eldest = sort { $age{$b} <=> $age{$a} } keys %age;

# sort using explicit subroutine name
sub byage {
    $age{$a} <=> $age{$b}; # presuming numeric
}
@sortedclass = sort byage @class;

sub backwards { $b cmp $a }
@harry = qw(dog cat x Cain Abel);
@george = qw(gone chased yz Punished Axed);
print sort @harry;
    # prints AbelCaincatdogx
print sort backwards @harry;
    # prints xdogcatCainAbel
print sort @george, 'to', @harry;
    # prints AbelAxedCainPunishedcatchaseddoggonetoxyz

# inefficiently sort by descending numeric compare using
# the first integer after the first = sign, or the
# whole record case-insensitively otherwise
@new = sort {
    ($b =~ /\=(\d+)/) [0] <=> ($a =~ /\=(\d+)/) [0]
    ||
    uc($a) cmp uc($b)
} @old;
```

```

# same thing, but much more efficiently;
# we'll build auxiliary indices instead
# for speed
@nums = @caps = ();
for (@old) {
    push @nums, /(\\d+)/;
    push @caps, uc($_);
}

@new = @old[ sort {
                    $nums[$b] <=> $nums[$a]
                    ||
                    $caps[$a] cmp $caps[$b]
                } 0..$#old
            ];

# same thing, but without any temps
@new = map { $_->[0] }
        sort { $b->[1] <=> $a->[1]
                ||
                $a->[2] cmp $b->[2]
            } map { [$_, /(\\d+)/, uc($_)] } @old;

# using a prototype allows you to use any comparison subroutine
# as a sort subroutine (including other package's subroutines)
package other;
sub backwards ($$) { $_[1] cmp $_[0]; }      # $a and $b are not set here

package main;
@new = sort other::backwards @old;

# guarantee stability, regardless of algorithm
use sort 'stable';
@new = sort { substr($a, 3, 5) cmp substr($b, 3, 5) } @old;

# force use of mergesort (not portable outside Perl 5.8)
use sort '_mergesort'; # note discouraging _
@new = sort { substr($a, 3, 5) cmp substr($b, 3, 5) } @old;

```

If you're using `strict`, you *must not* declare `$a` and `$b` as lexicals. They are package globals. That means if you're in the main package and type

```
@articles = sort { $b <=> $a } @files;
```

then `$a` and `$b` are `$main::a` and `$main::b` (or `$::a` and `$::b`), but if you're in the `FooPack` package, it's the same as typing

```
@articles = sort { $FooPack::b <=> $FooPack::a } @files;
```

The comparison function is required to behave. If it returns inconsistent results (sometimes saying `$x[1]` is less than `$x[2]` and sometimes saying the opposite, for example) the results are not well-defined.

splice ARRAY,OFFSET,LENGTH,LIST
splice ARRAY,OFFSET,LENGTH
splice ARRAY,OFFSET
splice ARRAY

Removes the elements designated by `OFFSET` and `LENGTH` from an array, and replaces them with the elements of `LIST`, if any. In list context, returns the elements removed from the array. In scalar context, returns the last element removed, or `undef` if no elements are removed. The

array grows or shrinks as necessary. If OFFSET is negative then it starts that far from the end of the array. If LENGTH is omitted, removes everything from OFFSET onward. If LENGTH is negative, removes the elements from OFFSET onward except for -LENGTH elements at the end of the array. If both OFFSET and LENGTH are omitted, removes everything. If OFFSET is past the end of the array, perl issues a warning, and splices at the end of the array.

The following equivalences hold (assuming `$[== 0`):

```
push(@a,$x,$y)      splice(@a,@a,0,$x,$y)
pop(@a)             splice(@a,-1)
shift(@a)           splice(@a,0,1)
unshift(@a,$x,$y)   splice(@a,0,0,$x,$y)
$a[$x] = $y         splice(@a,$x,1,$y)
```

Example, assuming array lengths are passed before arrays:

```
sub aeq {      # compare two list values
    my(@a) = splice(@_,0,shift);
    my(@b) = splice(@_,0,shift);
    return 0 unless @a == @b;      # same len?
    while (@a) {
        return 0 if pop(@a) ne pop(@b);
    }
    return 1;
}
if (&aeq($len,@foo[1..$len],0+@bar,@bar)) { ... }
```

split /PATTERN/EXPR,LIMIT

split /PATTERN/EXPR

split /PATTERN/

split Splits a string into a list of strings and returns that list. By default, empty leading fields are preserved, and empty trailing ones are deleted.

In scalar context, returns the number of fields found and splits into the `@_` array. Use of split in scalar context is deprecated, however, because it clobbers your subroutine arguments.

If EXPR is omitted, splits the `$_` string. If PATTERN is also omitted, splits on whitespace (after skipping any leading whitespace). Anything matching PATTERN is taken to be a delimiter separating the fields. (Note that the delimiter may be longer than one character.)

If LIMIT is specified and positive, it represents the maximum number of fields the EXPR will be split into, though the actual number of fields returned depends on the number of times PATTERN matches within EXPR. If LIMIT is unspecified or zero, trailing null fields are stripped (which potential users of pop would do well to remember). If LIMIT is negative, it is treated as if an arbitrarily large LIMIT had been specified. Note that splitting an EXPR that evaluates to the empty string always returns the empty list, regardless of the LIMIT specified.

A pattern matching the null string (not to be confused with a null pattern `//`, which is just one member of the set of patterns matching a null string) will split the value of EXPR into separate characters at each point it matches that way. For example:

```
print join(':', split(/ */, 'hi there'));
```

produces the output `'h:i:t:h:e:r:e'`.

Using the empty pattern `//` specifically matches the null string, and is not to be confused with the use of `//` to mean "the last successful pattern match".

Empty leading (or trailing) fields are produced when there are positive width matches at the beginning (or end) of the string; a zero-width match at the beginning (or end) of the string does not produce an empty field. For example:

```
print join(':', split(/(?=\w)/, 'hi there!'));
```

produces the output 'hi:t:h:e:r:e!'.

The LIMIT parameter can be used to split a line partially

```
($login, $passwd, $remainder) = split(/:/, $_, 3);
```

When assigning to a list, if LIMIT is omitted, Perl supplies a LIMIT one larger than the number of variables in the list, to avoid unnecessary work. For the list above LIMIT would have been 4 by default. In time critical applications it behooves you not to split into more fields than you really need.

If the PATTERN contains parentheses, additional list elements are created from each matching substring in the delimiter.

```
split(/[,-]/, "1-10,20", 3);
```

produces the list value

```
(1, '-', 10, ',', 20)
```

If you had the entire header of a normal Unix email message in \$header, you could split it up into fields and their values this way:

```
$header =~ s/\n\s+/ /g; # fix continuation lines
%hdrs   = (UNIX_FROM => split /^(\S*?):\s*/m, $header);
```

The pattern /PATTERN/ may be replaced with an expression to specify patterns that vary at runtime. (To do runtime compilation only once, use /\$variable/o.)

As a special case, specifying a PATTERN of space (' ') will split on white space just as split with no arguments does. Thus, split(' ') can be used to emulate **awk**'s default behavior, whereas split(/ /) will give you as many null initial fields as there are leading spaces. A split on /\s+/ is like a split(' ') except that any leading whitespace produces a null first field. A split with no arguments really does a split(' ', \$_) internally.

A PATTERN of /^/ is treated as if it were /^/m, since it isn't much use otherwise.

Example:

```
open(PASSWD, '/etc/passwd');
while (<PASSWD>) {
    chomp;
    ($login, $passwd, $uid, $gid,
     $gcos, $home, $shell) = split(/:/);
    #...
}
```

As with regular pattern matching, any capturing parentheses that are not matched in a split() will be set to undef when returned:

```
@fields = split /(A)|B/, "1A2B3";
# @fields is (1, 'A', 2, undef, 3)
```

sprintf FORMAT, LIST

Returns a string formatted by the usual printf conventions of the C library function sprintf. See below for more details and see [sprintf\(3\)](#) or [printf\(3\)](#) on your system for an explanation of the general principles.

For example:

```
# Format number with up to 8 leading zeroes
```



```
$result = sprintf("%08d", $number);

# Round number to 3 digits after decimal point
$rounded = sprintf("%.3f", $number);
```

Perl does its own `sprintf` formatting—it emulates the C function `sprintf`, but it doesn't use it (except for floating-point numbers, and even then only the standard modifiers are allowed). As a result, any non-standard extensions in your local `sprintf` are not available from Perl.

Unlike `printf`, `sprintf` does not do what you probably mean when you pass it an array as your first argument. The array is given scalar context, and instead of using the 0th element of the array as the format, Perl will use the count of elements in the array as the format, which is almost never useful.

Perl's `sprintf` permits the following universally-known conversions:

```
%%    a percent sign
%c    a character with the given number
%s    a string
%d    a signed integer, in decimal
%u    an unsigned integer, in decimal
%o    an unsigned integer, in octal
%x    an unsigned integer, in hexadecimal
%e    a floating-point number, in scientific notation
%f    a floating-point number, in fixed decimal notation
%g    a floating-point number, in %e or %f notation
```

In addition, Perl permits the following widely-supported conversions:

```
%X    like %x, but using upper-case letters
%E    like %e, but using an upper-case "E"
%G    like %g, but with an upper-case "E" (if applicable)
%b    an unsigned integer, in binary
%p    a pointer (outputs the Perl value's address in hexadecimal)
%n    special: *stores* the number of characters output so far
      into the next variable in the parameter list
```

Finally, for backward (and we do mean "backward") compatibility, Perl permits these unnecessary but widely-supported conversions:

```
%i    a synonym for %d
%D    a synonym for %ld
%U    a synonym for %lu
%O    a synonym for %lo
%F    a synonym for %f
```

Note that the number of exponent digits in the scientific notation produced by `%e`, `%E`, `%g` and `%G` for numbers with the modulus of the exponent less than 100 is system-dependent: it may be three or less (zero-padded as necessary). In other words, 1.23 times ten to the 99th may be either "1.23e99" or "1.23e099".

Between the `%` and the format letter, you may specify a number of additional attributes controlling the interpretation of the format. In order, these are:

format parameter index

An explicit format parameter index, such as `2$`. By default `sprintf` will format the next unused argument in the list, but this allows you to take the arguments out of order. Eg:

```
printf '%2$d %1$d', 12, 34;      # prints "34 12"
printf '%3$d %d %1$d', 1, 2, 3; # prints "3 1 1"
```

flags

one or more of:

- space prefix positive number with a space
- + prefix positive number with a plus sign
- left-justify within the field
- 0 use zeros, not spaces, to right-justify
- # prefix non-zero octal with "0", non-zero hex with "0x", non-zero binary with "0b"

For example:

```
printf '<% d>', 12;    # prints "< 12>"
printf '<+%d>', 12;    # prints "<+12>"
printf '<%6s>', 12;    # prints "<      12>"
printf '<%-6s>', 12;    # prints "<12      >"
printf '<%06s>', 12;    # prints "<000012>"
printf '<%#x>', 12;     # prints "<0xc>"
```

vector flag

The vector flag `v`, optionally specifying the join string to use. This flag tells perl to interpret the supplied string as a vector of integers, one for each character in the string, separated by a given string (a dot `.` by default). This can be useful for displaying ordinal values of characters in arbitrary strings:

```
printf "version is v%vd\n", $^V;    # Perl's version
```

Put an asterisk `*` before the `v` to override the string to use to separate the numbers:

```
printf "address is %*vX\n", ":", $addr;    # IPv6 address
printf "bits are %0*v8b\n", " ", $bits;    # random bitstring
```

You can also explicitly specify the argument number to use for the join string using `eg *2$v`:

```
printf '%*4$vX %*4$vX %*4$vX', @addr[1..3], ":";    # 3 IPv6 addresses
```

(minimum) width

Arguments are usually formatted to be only as wide as required to display the given value. You can override the width by putting a number here, or get the width from the next argument (with `*`) or from a specified argument (with eg `*2$`) :

```
printf '<%s>', "a";    # prints "<a>"
printf '<%6s>', "a";    # prints "<      a>"
printf '<%*s>', 6, "a";  # prints "<      a>"
printf '<%*2$s>', "a", 6; # prints "<      a>"
printf '<%2s>', "long";  # prints "<long>" (does not truncate)
```

If a field width obtained through `*` is negative, it has the same effect as the `–` flag: left-justification.

precision, or maximum width

You can specify a precision (for numeric conversions) or a maximum width (for string conversions) by specifying a `.` followed by a number. For floating point formats, this specifies the number of decimal places to show (the default being 6), eg:

```
# these examples are subject to system-specific variation
printf '<%f>', 1;    # prints "<1.000000>"
printf '<%.1f>', 1;  # prints "<1.0>"
printf '<%.0f>', 1;  # prints "<1>"
printf '<%e>', 10;   # prints "<1.000000e+01>"
```

```
printf '<%.1e>', 10; # prints "<1.0e+01>"
```

For integer conversions, specifying a precision implies that the output of the number itself should be zero-padded to this width:

```
printf '<%.6x>', 1;          # prints "<000001>"
printf '<%#.6x>', 1;         # prints "<0x000001>"
printf '<%-10.6x>', 1;      # prints "<000001      >"
```

For string conversions, specifying a precision truncates the string to fit in the specified width:

```
printf '<%.5s>', "truncated"; # prints "<trunc>"
printf '<%10.5s>', "truncated"; # prints "<      trunc>"
```

You can also get the precision from the next argument using `.*`:

```
printf '<%.6x>', 1;          # prints "<000001>"
printf '<%.*x>', 6, 1;       # prints "<000001>"
```

You cannot currently get the precision from a specified number, but it is intended that this will be possible in the future using eg `.*2$`:

```
printf '<%.*2$x>', 1, 6;    # INVALID, but in future will print "<000001"
```

size For numeric conversions, you can specify the size to interpret the number as using `l`, `h`, `V`, `q`, `L` or `ll`. For integer conversions, numbers are usually assumed to be whatever the default integer size is on your platform (usually 32 or 64 bits), but you can override this to use instead one of the standard C types, as supported by the compiler used to build Perl:

<code>l</code>	interpret integer as C type "long" or "unsigned long"
<code>h</code>	interpret integer as C type "short" or "unsigned short"
<code>q</code> , <code>L</code> or <code>ll</code>	interpret integer as C type "long long" or "unsigned long long" (if your platform supports such a type, else it is an error)

For floating point conversions, numbers are usually assumed to be the default floating point size on your platform (double or long double), but you can force 'long double' with `q`, `L` or `ll` if your platform supports them.

The size specifier '`V`' has no effect for Perl code, but it is supported for compatibility with XS code; it means 'use the standard size for a Perl integer (or floating-point number)', which is already the default for Perl code.

order of arguments

Normally, `sprintf` takes the next unused argument as the value to format for each format specification. If the format specification uses `*` to require additional arguments, these are consumed from the argument list in the order in which they appear in the format specification *before* the value to format. Where an argument is specified using an explicit index, this does not affect the normal order for the arguments (even when the explicitly specified index would have been the next argument in any case).

So:

```
printf '<%*.*s>', $a, $b, $c;
```

would use `$a` for the width, `$b` for the precision and `$c` as the value to format, while:

```
print '<%*1$.*s>', $a, $b;
```

would use `$a` for the width and the precision, and `$b` as the value to format.

Here are some more examples – beware that when using an explicit index, the `$` may need to be escaped:

```
printf "%2\${d} %d\n",    12, 34# will print "34 12\n"
printf "%2\${d} %d %d\n", 12, 34;      # will print "34 12 34\n"
printf "%3\${d} %d %d\n", 12, 34, 56;  # will print "56 12 34\n"
printf "%2\${*3}\${d} %d\n", 12, 34, 3; # will print " 34 12\n"
```

If use `locale` is in effect, the character used for the decimal point in formatted real numbers is affected by the `LC_NUMERIC` locale. See [perllocale](#).

If Perl understands "quads" (64-bit integers) (this requires either that the platform natively support quads or that Perl be specifically compiled to support quads), the characters

```
d u o x X b i D U O
```

print quads, and they may optionally be preceded by

```
l l L q
```

For example

```
%lld %16LX %qo
```

You can find out whether your Perl supports quads via [Config](#):

```
use Config;
($Config{use64bitint} eq 'define' || $Config{longsize} == 8) &&
    print "quads\n";
```

If Perl understands "long doubles" (this requires that the platform support long doubles), the flags

```
e f g E F G
```

may optionally be preceded by

```
l l L
```

For example

```
%llf %Lg
```

You can find out whether your Perl supports long doubles via [Config](#):

```
use Config;
$Config{d_longdbl} eq 'define' && print "long doubles\n";
```

sqrt EXPR

sqrt Return the square root of EXPR. If EXPR is omitted, returns square root of `$_`. Only works on non-negative operands, unless you've loaded the standard `Math::Complex` module.

```
use Math::Complex;
print sqrt(-2);    # prints 1.4142135623731i
```

srand EXPR

srand Sets the random number seed for the `rand` operator.

The point of the function is to "seed" the `rand` function so that `rand` can produce a different sequence each time you run your program.

If `srand()` is not called explicitly, it is called implicitly at the first use of the `rand` operator. However, this was not the case in versions of Perl before 5.004, so if your script will run under older Perl versions, it should call `srand`.

Most programs won't even call `srand()` at all, except those that need a cryptographically-strong starting point rather than the generally acceptable default, which is based on time of day, process ID, and memory allocation, or the `/dev/urandom` device, if available.

You can call `srand($seed)` with the same `$seed` to reproduce the *same* sequence from `rand()`, but this is usually reserved for generating predictable results for testing or debugging. Otherwise, don't call `srand()` more than once in your program.

Do **not** call `srand()` (i.e. without an argument) more than once in a script. The internal state of the random number generator should contain more entropy than can be provided by any seed, so calling `srand()` again actually *loses* randomness.

Most implementations of `srand` take an integer and will silently truncate decimal numbers. This means `srand(42)` will usually produce the same results as `srand(42.1)`. To be safe, always pass `srand` an integer.

In versions of Perl prior to 5.004 the default seed was just the current time. This isn't a particularly good seed, so many old programs supply their own seed value (often `time ^ $$` or `time ^ ($$ + ($$ < 15))`), but that isn't necessary any more.

Note that you need something much more random than the default seed for cryptographic purposes. Checksumming the compressed output of one or more rapidly changing operating system status programs is the usual method. For example:

```
srand (time ^ $$ ^ unpack "%L*", 'ps axww | gzip');
```

If you're particularly concerned with this, see the `Math::TrulyRandom` module in CPAN.

Frequently called programs (like CGI scripts) that simply use

```
time ^ $$
```

for a seed can fall prey to the mathematical property that

$$a^b == (a+1)^{(b+1)}$$

one-third of the time. So don't do that.

stat FILEHANDLE

stat EXPR

stat Returns a 13-element list giving the status info for a file, either the file opened via **FILEHANDLE**, or named by **EXPR**. If **EXPR** is omitted, it stats `$_`. Returns a null list if the **stat** fails. Typically used as follows:

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
 $atime,$mtime,$ctime,$blksize,$blocks)
= stat($filename);
```

Not all fields are supported on all filesystem types. Here are the meaning of the fields:

0 dev	device number of filesystem
1 ino	inode number
2 mode	file mode (type and permissions)
3 nlink	number of (hard) links to the file
4 uid	numeric user ID of file's owner
5 gid	numeric group ID of file's owner
6 rdev	the device identifier (special files only)
7 size	total size of file, in bytes
8 atime	last access time in seconds since the epoch
9 mtime	last modify time in seconds since the epoch
10 ctime	inode change time in seconds since the epoch (*)
11 blksize	preferred block size for file system I/O
12 blocks	actual number of blocks allocated

(The epoch was at 00:00 January 1, 1970 GMT.)

(*) The ctime field is non-portable, in particular you cannot expect it to be a "creation time", see *Files and Filesystems in perlport* for details.

If stat is passed the special filehandle consisting of an underline, no stat is done, but the current contents of the stat structure from the last stat or filetest are returned. Example:

```
if (-x $file && (($d) = stat(_)) && $d < 0) {
    print "$file is executable NFS file\n";
}
```

(This works on machines only for which the device number is negative under NFS.)

Because the mode contains both the file type and its permissions, you should mask off the file type portion and (s)printf using a "%o" if you want to see the real permissions.

```
$mode = (stat($filename))[2];
printf "Permissions are %04o\n", $mode & 07777;
```

In scalar context, stat returns a boolean value indicating success or failure, and, if successful, sets the information associated with the special filehandle `_`.

The File::stat module provides a convenient, by-name access mechanism:

```
use File::stat;
$sb = stat($filename);
printf "File is %s, size is %s, perm %04o, mtime %s\n",
    $filename, $sb->size, $sb->mode & 07777,
    scalar localtime $sb->mtime;
```

You can import symbolic mode constants (S_IF*) and functions (S_IS*) from the Fcntl module:

```
use Fcntl ':mode';
$mode = (stat($filename))[2];
$user_rwx      = ($mode & S_IRWXU) >> 6;
$group_read    = ($mode & S_IRGRP) >> 3;
$other_execute = $mode & S_IXOTH;
printf "Permissions are %04o\n", S_ISMODE($mode), "\n";
$is_setuid     = $mode & S_ISUID;
$is_setgid     = S_ISDIR($mode);
```

You could write the last two using the `-u` and `-d` operators. The commonly available S_IF* constants are

```
# Permissions: read, write, execute, for user, group, others.
S_IRWXU S_IRUSR S_IWUSR S_IXUSR
S_IRWXG S_IRGRP S_IWGRP S_IXGRP
S_IRWXO S_IROTH S_IWOTH S_IXOTH

# Setuid/Setgid/Stickiness.
S_ISUID S_ISGID S_ISVTX S_ISTXT

# File types. Not necessarily all are available on your system.
S_IFREG S_IFDIR S_IFLNK S_IFBLK S_ISCHR S_IFIFO S_IFSOCK S_IFWHT S_ENFMT

# The following are compatibility aliases for S_IRUSR, S_IWUSR, S_IXUSR.
S_IREAD S_IWRITE S_IEXEC
```

and the S_IF* functions are

```

S_IFMODE($mode)      the part of $mode containing the permission bits
                      and the setuid/setgid/sticky bits

S_IFMT($mode)        the part of $mode containing the file type
                      which can be bit-anded with e.g. S_IFREG
                      or with the following functions

# The operators -f, -d, -l, -b, -c, -p, and -s.

S_ISREG($mode) S_ISDIR($mode) S_ISLNK($mode)
S_ISBLK($mode) S_ISCHR($mode) S_ISFIFO($mode) S_ISSOCK($mode)

# No direct -X operator counterpart, but for the first one
# the -g operator is often equivalent. The ENFMT stands for
# record flocking enforcement, a platform-dependent feature.

S_ISENFMT($mode) S_ISWHT($mode)

```

See your native `chmod(2)` and `stat(2)` documentation for more details about the S_* constants.

study SCALAR

study Takes extra time to study SCALAR (\$_ if unspecified) in anticipation of doing many pattern matches on the string before it is next modified. This may or may not save time, depending on the nature and number of patterns you are searching on, and on the distribution of character frequencies in the string to be searched—you probably want to compare run times with and without it to see which runs faster. Those loops which scan for many short constant strings (including the constant parts of more complex patterns) will benefit most. You may have only one `study` active at a time—if you study a different scalar the first is "unstudied". (The way `study` works is this: a linked list of every character in the string to be searched is made, so we know, for example, where all the 'k' characters are. From each search string, the rarest character is selected, based on some static frequency tables constructed from some C programs and English text. Only those places that contain this "rarest" character are examined.)

For example, here is a loop that inserts index producing entries before any line containing a certain pattern:

```

while (<>) {
    study;
    print ".IX foo\n"      if /\bfoo\b/;
    print ".IX bar\n"      if /\bbar\b/;
    print ".IX blurfl\n"   if /\bblurfl\b/;
    # ...
    print;
}

```

In searching for `/\bfoo\b/`, only those locations in `$_` that contain `f` will be looked at, because `f` is rarer than `o`. In general, this is a big win except in pathological cases. The only question is whether it saves you more time than it took to build the linked list in the first place.

Note that if you have to look for strings that you don't know till runtime, you can build an entire loop as a string and `eval` that to avoid recompiling all your patterns all the time. Together with undefining `$/` to input entire files as one record, this can be very fast, often faster than specialized programs like `fgrep(1)`. The following scans a list of files (`@files`) for a list of words (`@words`), and prints out the names of those files that contain a match:

```

$search = 'while (<>) { study;';
foreach $word (@words) {
    $search .= "++\${seen}{\${ARGV} if /\b$word\b/;\n";
}

```

```

$search .= "}";
@ARGV = @files;
undef $/;
eval $search;          # this screams
$/ = "\n";             # put back to normal input delimiter
foreach $file (sort keys(%seen)) {
    print $file, "\n";
}

```

sub NAME BLOCK

sub NAME (PROTO) BLOCK

sub NAME : ATTRS BLOCK

sub NAME (PROTO) : ATTRS BLOCK

This is subroutine definition, not a real function *per se*. Without a BLOCK it's just a forward declaration. Without a NAME, it's an anonymous function declaration, and does actually return a value: the CODE ref of the closure you just created.

See [perlsub](#) and [perlref](#) for details about subroutines and references, and [attributes](#) and [Attribute::Handlers](#) for more information about attributes.

substr EXPR,OFFSET,LENGTH,REPLACEMENT

substr EXPR,OFFSET,LENGTH

substr EXPR,OFFSET

Extracts a substring out of EXPR and returns it. First character is at offset , or whatever you've set \$[to (but don't do that). If OFFSET is negative (or more precisely, less than \$[), starts that far from the end of the string. If LENGTH is omitted, returns everything to the end of the string. If LENGTH is negative, leaves that many characters off the end of the string.

You can use the substr() function as an lvalue, in which case EXPR must itself be an lvalue. If you assign something shorter than LENGTH, the string will shrink, and if you assign something longer than LENGTH, the string will grow to accommodate it. To keep the string the same length you may need to pad or chop your value using sprintf.

If OFFSET and LENGTH specify a substring that is partly outside the string, only the part within the string is returned. If the substring is beyond either end of the string, substr() returns the undefined value and produces a warning. When used as an lvalue, specifying a substring that is entirely outside the string is a fatal error. Here's an example showing the behavior for boundary cases:

```

my $name = 'fred';
substr($name, 4) = 'dy';          # $name is now 'freddy'
my $null = substr $name, 6, 2;    # returns '' (no warning)
my $oops = substr $name, 7;      # returns undef, with warning
substr($name, 7) = 'gap';        # fatal error

```

An alternative to using substr() as an lvalue is to specify the replacement string as the 4th argument. This allows you to replace parts of the EXPR and return what was there before in one operation, just as you can with splice().

symlink OLDFILE,NEWFILE

Creates a new filename symbolically linked to the old filename. Returns 1 for success, otherwise. On systems that don't support symbolic links, produces a fatal error at run time. To check for that, use eval:

```
$symlink_exists = eval { symlink("", ""); 1 };
```

syscall LIST

Calls the system call specified as the first element of the list, passing the remaining elements as arguments to the system call. If unimplemented, produces a fatal error. The arguments are

interpreted as follows: if a given argument is numeric, the argument is passed as an int. If not, the pointer to the string value is passed. You are responsible to make sure a string is pre-extended long enough to receive any result that might be written into a string. You can't use a string literal (or other read-only string) as an argument to `syscall` because Perl has to assume that any string pointer might be written through. If your integer arguments are not literals and have never been interpreted in a numeric context, you may need to add to them to force them to look like numbers. This emulates the `syswrite` function (or vice versa):

```
require 'syscall.ph';           # may need to run h2ph
$s = "hi there\n";
syscall(&SYS_write, fileno(STDOUT), $s, length $s);
```

Note that Perl supports passing of up to only 14 arguments to your system call, which in practice should usually suffice.

`syscall` returns whatever value returned by the system call it calls. If the system call fails, `syscall` returns `-1` and sets `$!` (`errno`). Note that some system calls can legitimately return `-1`. The proper way to handle such calls is to assign `$!=0`; before the call and check the value of `$!` if `syscall` returns `-1`.

There's a problem with `syscall(&SYS_pipe)`: it returns the file number of the read end of the pipe it creates. There is no way to retrieve the file number of the other end. You can avoid this problem by using `pipe` instead.

`sysopen FILEHANDLE,FILENAME,MODE` `sysopen FILEHANDLE,FILENAME,MODE,PERMS`

Opens the file whose filename is given by `FILENAME`, and associates it with `FILEHANDLE`. If `FILEHANDLE` is an expression, its value is used as the name of the real filehandle wanted. This function calls the underlying operating system's `open` function with the parameters `FILENAME`, `MODE`, `PERMS`.

The possible values and flag bits of the `MODE` parameter are system-dependent; they are available via the standard module `Fcntl`. See the documentation of your operating system's `open` to see which values and flag bits are available. You may combine several flags using the `|`-operator.

Some of the most common values are `O_RDONLY` for opening the file in read-only mode, `O_WRONLY` for opening the file in write-only mode, and `O_RDWR` for opening the file in read-write mode, and.

For historical reasons, some values work on almost every system supported by perl: zero means read-only, one means write-only, and two means read/write. We know that these values do *not* work under OS/390 & VM/ESA Unix and on the Macintosh; you probably don't want to use them in new code.

If the file named by `FILENAME` does not exist and the `open` call creates it (typically because `MODE` includes the `O_CREAT` flag), then the value of `PERMS` specifies the permissions of the newly created file. If you omit the `PERMS` argument to `sysopen`, Perl uses the octal value `0666`. These permission values need to be in octal, and are modified by your process's current `umask`.

In many systems the `O_EXCL` flag is available for opening files in exclusive mode. This is **not** locking: exclusiveness means here that if the file already exists, `sysopen()` fails. The `O_EXCL` wins `O_TRUNC`.

Sometimes you may want to truncate an already-existing file: `O_TRUNC`.

You should seldom if ever use `0644` as argument to `sysopen`, because that takes away the user's option to have a more permissive `umask`. Better to omit it. See the `perlfunc(1)` entry on `umask` for more on this.

Note that `sysopen` depends on the `fdopen()` C library function. On many UNIX systems, `fdopen()` is known to fail when file descriptors exceed a certain value, typically 255. If you need more file descriptors than that, consider rebuilding Perl to use the `sfio` library, or perhaps using the `POSIX::open()` function.

See [perlopentut](#) for a kinder, gentler explanation of opening files.

`sysread FILEHANDLE, SCALAR, LENGTH, OFFSET`

`sysread FILEHANDLE, SCALAR, LENGTH`

Attempts to read `LENGTH` *characters* of data into variable `SCALAR` from the specified `FILEHANDLE`, using the system call `read(2)`. It bypasses buffered IO, so mixing this with other kinds of reads, `print`, `write`, `seek`, `tell`, or `eof` can cause confusion because `stdio` usually buffers data. Returns the number of characters actually read, at end of file, or `undef` if there was an error. `SCALAR` will be grown or shrunk so that the last byte actually read is the last byte of the scalar after the read.

Note the *characters*: depending on the status of the filehandle, either (8-bit) bytes or characters are read. By default all filehandles operate on bytes, but for example if the filehandle has been opened with the `:utf8` I/O layer (see [/open](#), and the `open` pragma, [open](#)), the I/O will operate on characters, not bytes.

An `OFFSET` may be specified to place the read data at some place in the string other than the beginning. A negative `OFFSET` specifies placement at that many characters counting backwards from the end of the string. A positive `OFFSET` greater than the length of `SCALAR` results in the string being padded to the required size with `"\0"` bytes before the result of the read is appended.

There is no `syseof()` function, which is ok, since `eof()` doesn't work very well on device files (like `ttys`) anyway. Use `sysread()` and check for a return value for 0 to decide whether you're done.

`sysseek FILEHANDLE, POSITION, WHENCE`

Sets `FILEHANDLE`'s system position *in bytes* using the system call `lseek(2)`. `FILEHANDLE` may be an expression whose value gives the name of the filehandle. The values for `WHENCE` are to set the new position to `POSITION`, 1 to set it to the current position plus `POSITION`, and 2 to set it to EOF plus `POSITION` (typically negative).

Note the *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `:utf8` I/O layer), `tell()` will return byte offsets, not character offsets (because implementing that would render `sysseek()` very slow).

`sysseek()` bypasses normal buffered io, so mixing this with reads (other than `sysread`, for example `><` or `read()`) `print`, `write`, `seek`, `tell`, or `eof` may cause confusion.

For `WHENCE`, you may also use the constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` (start of the file, current position, end of the file) from the `Fcntl` module. Use of the constants is also more portable than relying on 0, 1, and 2. For example to define a "systell" function:

```
use Fcntl 'SEEK_CUR';
sub systell { sysseek($_[0], 0, SEEK_CUR) }
```

Returns the new position, or the undefined value on failure. A position of zero is returned as the string `"0 but true"`; thus `sysseek` returns true on success and false on failure, yet you can still easily determine the new position.

`system LIST`

`system PROGRAM LIST`

Does exactly the same thing as `exec LIST`, except that a fork is done first, and the parent process waits for the child process to complete. Note that argument processing varies depending on the number of arguments. If there is more than one argument in `LIST`, or if `LIST` is an array

with more than one value, starts the program given by the first element of the list with arguments given by the rest of the list. If there is only one scalar argument, the argument is checked for shell metacharacters, and if there are any, the entire argument is passed to the system's command shell for parsing (this is `/bin/sh -c` on Unix platforms, but varies on other platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to `execvp`, which is more efficient.

Beginning with v5.6.0, Perl will attempt to flush all files opened for output before any operation that may do a fork, but this may not be supported on some platforms (see [perlport](#)). To be safe, you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush()` method of `IO::Handle` on any open handles.

The return value is the exit status of the program as returned by the `wait` call. To get the actual exit value shift right by eight (see below). See also [/exec](#). This is *not* what you want to use to capture the output from a command, for that you should use merely backticks or `qx//`, as described in [‘STRING’ in perllop](#). Return value of `-1` indicates a failure to start the program (inspect `$!` for the reason).

Like `exec`, `system` allows you to lie to a program about its name if you use the `system PROGRAM LIST` syntax. Again, see [/exec](#).

Because `system` and backticks block `SIGINT` and `SIGQUIT`, killing the program they're running doesn't actually interrupt your program.

```
@args = ("command", "arg1", "arg2");
system(@args) == 0
    or die "system @args failed: $?"
```

You can check all the failure possibilities by inspecting `$?` like this:

```
$exit_value = $? >> 8;
$signal_num = $? & 127;
$dumped_core = $? & 128;
```

or more portably by using the `W*`() calls of the POSIX extension; see [perlport](#) for more information.

When the arguments get executed via the system shell, results and return codes will be subject to its quirks and capabilities. See [‘STRING’ in perllop](#) and [/exec](#) for details.

syswrite FILEHANDLE, SCALAR, LENGTH, OFFSET

syswrite FILEHANDLE, SCALAR, LENGTH

syswrite FILEHANDLE, SCALAR

Attempts to write `LENGTH` characters of data from variable `SCALAR` to the specified `FILEHANDLE`, using the system call `write(2)`. If `LENGTH` is not specified, writes whole `SCALAR`. It bypasses buffered IO, so mixing this with reads (other than `sysread()`), `print`, `write`, `seek`, `tell`, or `eof` may cause confusion because `stdio` usually buffers data. Returns the number of characters actually written, or `undef` if there was an error. If the `LENGTH` is greater than the available data in the `SCALAR` after the `OFFSET`, only as much data as is available will be written.

An `OFFSET` may be specified to write the data from some part of the string other than the beginning. A negative `OFFSET` specifies writing that many characters counting backwards from the end of the string. In the case the `SCALAR` is empty you can use `OFFSET` but only zero offset.

Note the *characters*: depending on the status of the filehandle, either (8-bit) bytes or characters are written. By default all filehandles operate on bytes, but for example if the filehandle has been opened with the `:utf8` I/O layer (see [/open](#), and the open pragma, [open](#)), the I/O will operate on characters, not bytes.

tell FILEHANDLE

tell Returns the current position *in bytes* for FILEHANDLE, or `-1` on error. FILEHANDLE may be an expression whose value gives the name of the actual filehandle. If FILEHANDLE is omitted, assumes the file last read.

Note the *in bytes*: even if the filehandle has been set to operate on characters (for example by using the `:utf8` open layer), `tell()` will return byte offsets, not character offsets (because that would render `seek()` and `tell()` rather slow).

The return value of `tell()` for the standard streams like the STDIN depends on the operating system: it may return `-1` or something else. `tell()` on pipes, fifos, and sockets usually returns `-1`.

There is no `sysstell` function. Use `sysseek(FH, 0, 1)` for that.

Do not use `tell()` on a filehandle that has been opened using `sysopen()`, use `sysseek()` for that as described above. Why? Because `sysopen()` creates unbuffered, "raw", filehandles, while `open()` creates buffered filehandles. `sysseek()` make sense only on the first kind, `tell()` only makes sense on the second kind.

telldir DIRHANDLE

Returns the current position of the `readdir` routines on DIRHANDLE. Value may be given to `seekdir` to access a particular location in a directory. Has the same caveats about possible directory compaction as the corresponding system library routine.

tie VARIABLE, CLASSNAME, LIST

This function binds a variable to a package class that will provide the implementation for the variable. VARIABLE is the name of the variable to be enchanted. CLASSNAME is the name of a class implementing objects of correct type. Any additional arguments are passed to the new method of the class (meaning `TIESCALAR`, `TIEHANDLE`, `TIEARRAY`, or `TIEHASH`). Typically these are arguments such as might be passed to the `dbm_open()` function of C. The object returned by the new method is also returned by the `tie` function, which would be useful if you want to access other methods in CLASSNAME.

Note that functions such as `keys` and `values` may return huge lists when used on large objects, like DBM files. You may prefer to use the `each` function to iterate over such. Example:

```
# print out history file offsets
use NDBM_File;
tie(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

A class implementing a hash should have the following methods:

```
TIEHASH classname, LIST
FETCH this, key
STORE this, key, value
DELETE this, key
CLEAR this
EXISTS this, key
FIRSTKEY this
NEXTKEY this, lastkey
DESTROY this
UNTIE this
```

A class implementing an ordinary array should have the following methods:

```
TIEARRAY classname, LIST
FETCH this, key
STORE this, key, value
FETCHSIZE this
STORESIZE this, count
CLEAR this
PUSH this, LIST
POP this
SHIFT this
UNSHIFT this, LIST
SPLICE this, offset, length, LIST
EXTEND this, count
DESTROY this
UNTIE this
```

A class implementing a file handle should have the following methods:

```
TIEHANDLE classname, LIST
READ this, scalar, length, offset
READLINE this
GETC this
WRITE this, scalar, length, offset
PRINT this, LIST
PRINTF this, format, LIST
BINMODE this
EOF this
FILENO this
SEEK this, position, whence
TELL this
OPEN this, mode, LIST
CLOSE this
DESTROY this
UNTIE this
```

A class implementing a scalar should have the following methods:

```
TIESCALAR classname, LIST
FETCH this,
STORE this, value
DESTROY this
UNTIE this
```

Not all methods indicated above need be implemented. See [perltie](#), [Tie::Hash](#), [Tie::Array](#), [Tie::Scalar](#), and [Tie::Handle](#).

Unlike `dbmopen`, the `tie` function will not use or require a module for you—you need to do that explicitly yourself. See [DB_File](#) or the **Config** module for interesting `tie` implementations.

For further details see [perltie](#), "[tied VARIABLE](#)".

tied VARIABLE

Returns a reference to the object underlying VARIABLE (the same value that was originally returned by the `tie` call that bound the variable to a package.) Returns the undefined value if VARIABLE isn't tied to a package.

time Returns the number of non-leap seconds since whatever time the system considers to be the epoch (that's 00:00:00, January 1, 1904 for Mac OS, and 00:00:00 UTC, January 1, 1970 for most other systems). Suitable for feeding to `gmtime` and `localtime`.

For measuring time in better granularity than one second, you may use either the `Time::HiRes` module from CPAN, or if you have `gettimeofday(2)`, you may be able to use the `syscall` interface of Perl, see [perlfaq8](#) for details.

times Returns a four-element list giving the user and system times, in seconds, for this process and the children of this process.

```
($user, $system, $cuser, $csystem) = times;
```

In scalar context, `times` returns `$user`.

tr/// The transliteration operator. Same as `y///`. See [perlop](#).

truncate FILEHANDLE,LENGTH

truncate EXPR,LENGTH

Truncates the file opened on `FILEHANDLE`, or named by `EXPR`, to the specified length. Produces a fatal error if `truncate` isn't implemented on your system. Returns true if successful, the undefined value otherwise.

The behavior is undefined if `LENGTH` is greater than the length of the file.

uc EXPR

uc Returns an uppercased version of `EXPR`. This is the internal function implementing the `\U` escape in double-quoted strings. Respects current `LC_CTYPE` locale if `use locale` in force. See [perllocale](#) and [perlunicode](#) for more details about locale and Unicode support. It does not attempt to do titlecase mapping on initial letters. See `ucfirst` for that.

If `EXPR` is omitted, uses `$_`.

ucfirst EXPR

ucfirst Returns the value of `EXPR` with the first character in uppercase (titlecase in Unicode). This is the internal function implementing the `\u` escape in double-quoted strings. Respects current `LC_CTYPE` locale if `use locale` in force. See [perllocale](#) and [perlunicode](#) for more details about locale and Unicode support.

If `EXPR` is omitted, uses `$_`.

umask EXPR

umask Sets the umask for the process to `EXPR` and returns the previous value. If `EXPR` is omitted, merely returns the current umask.

The Unix permission `rwxr-x--` is represented as three sets of three bits, or three octal digits: `0750` (the leading 0 indicates octal and isn't one of the digits). The umask value is such a number representing disabled permissions bits. The permission (or "mode") values you pass `mkdir` or `sysopen` are modified by your umask, so even if you tell `sysopen` to create a file with permissions `0777`, if your umask is `0022` then the file will actually be created with permissions `0755`. If your umask were `0027` (group can't write; others can't read, write, or execute), then passing `sysopen 0666` would create a file with mode `0640` (`0666 & ~ 027` is `0640`).

Here's some advice: supply a creation mode of `0666` for regular files (in `sysopen`) and one of `0777` for directories (in `mkdir`) and executable files. This gives users the freedom of choice: if they want protected files, they might choose process umasks of `022`, `027`, or even the particularly antisocial mask of `077`. Programs should rarely if ever make policy decisions better left to the user. The exception to this is when writing files that should be kept private: mail files, web browser cookies, `.rhosts` files, and so on.

If `umask(2)` is not implemented on your system and you are trying to restrict access for *yourself* (i.e., `(EXPR & 0700) 0`), produces a fatal error at run time. If `umask(2)` is not implemented and you are not trying to restrict access for yourself, returns `undef`.

Remember that a `umask` is a number, usually given in octal; it is *not* a string of octal digits. See also [/oct](#), if all you have is a string.

undef EXPR

undef Undefines the value of `EXPR`, which must be an lvalue. Use only on a scalar value, an array (using `@`), a hash (using `%`), a subroutine (using `&`), or a typeglob (using `<*`). (Saying `undef $hash{$key}` will probably not do what you expect on most predefined variables or DBM list values, so don't do that; see [delete](#).) Always returns the undefined value. You can omit the `EXPR`, in which case nothing is undefined, but you still get an undefined value that you could, for instance, return from a subroutine, assign to a variable or pass as a parameter. Examples:

```
undef $foo;
undef $bar{'blurfl'};      # Compare to: delete $bar{'blurfl'};
undef @ary;
undef %hash;
undef &mysub;
undef *xyz;                # destroys $xyz, @xyz, %xyz, &xyz, etc.
return (wantarray ? (undef, $errmsg) : undef) if $they_blew_it;
select undef, undef, undef, 0.25;
($a, $b, undef, $c) = &foo; # Ignore third value returned
```

Note that this is a unary operator, not a list operator.

unlink LIST

unlink Deletes a list of files. Returns the number of files successfully deleted.

```
$cnt = unlink 'a', 'b', 'c';
unlink @goners;
unlink <*.bak>;
```

Note: `unlink` will not delete directories unless you are superuser and the `-U` flag is supplied to Perl. Even if these conditions are met, be warned that unlinking a directory can inflict damage on your filesystem. Use `rmdir` instead.

If `LIST` is omitted, uses `$_`.

unpack TEMPLATE,EXPR

`unpack` does the reverse of `pack`: it takes a string and expands it out into a list of values. (In scalar context, it returns merely the first value produced.)

The string is broken into chunks described by the `TEMPLATE`. Each chunk is converted separately to a value. Typically, either the string is a result of `pack`, or the bytes of the string represent a C structure of some kind.

The `TEMPLATE` has the same format as in the `pack` function. Here's a subroutine that does substring:

```
sub substr {
    my ($what, $where, $howmuch) = @_ ;
    unpack("x$where a$howmuch", $what);
}
```

and then there's

```
sub ordinal { unpack("c", $_[0]); } # same as ord()
```

In addition to fields allowed in `pack()`, you may prefix a field with a `%<number>` to indicate that you want a `<number>`-bit checksum of the items instead of the items themselves. Default is a

16-bit checksum. Checksum is calculated by summing numeric values of expanded values (for string fields the sum of `ord($char)` is taken, for bit fields the sum of zeroes and ones).

For example, the following computes the same number as the System V `sum` program:

```
$checksum = do {
    local $/; # slurp!
    unpack("%32C*",<>) % 65535;
};
```

The following efficiently counts the number of set bits in a bit vector:

```
$setbits = unpack("%32b*", $selectmask);
```

The `p` and `P` formats should be used with care. Since Perl has no way of checking whether the value passed to `unpack()` corresponds to a valid memory location, passing a pointer value that's not known to be valid is likely to have disastrous consequences.

If the repeat count of a field is larger than what the remainder of the input string allows, repeat count is decreased. If the input string is longer than one described by the `TEMPLATE`, the rest is ignored.

See [/pack](#) for more examples and notes.

untie VARIABLE

Breaks the binding between a variable and a package. (See `tie`.) Has no effect if the variable is not tied.

unshift ARRAY,LIST

Does the opposite of a `shift`. Or the opposite of a `push`, depending on how you look at it. Prepends list to the front of the array, and returns the new number of elements in the array.

```
unshift(@ARGV, '-e') unless $ARGV[0] =~ /^-/;
```

Note the `LIST` is prepended whole, not one element at a time, so the prepended elements stay in the same order. Use `reverse` to do the reverse.

use Module VERSION LIST

use Module VERSION

use Module LIST

use Module

use VERSION

Imports some semantics into the current package from the named module, generally by aliasing certain subroutine or variable names into your package. It is exactly equivalent to

```
BEGIN { require Module; import Module LIST; }
```

except that `Module` *must* be a bareword.

`VERSION` may be either a numeric argument such as `5.006`, which will be compared to `$]`, or a literal of the form `v5.6.1`, which will be compared to `^V` (aka `$PERL_VERSION`). A fatal error is produced if `VERSION` is greater than the version of the current Perl interpreter; Perl will not attempt to parse the rest of the file. Compare with [/require](#), which can do a similar check at run time.

Specifying `VERSION` as a literal of the form `v5.6.1` should generally be avoided, because it leads to misleading error messages under earlier versions of Perl which do not support this syntax. The equivalent numeric version should be used instead.

```
use v5.6.1;           # compile time version check
use 5.6.1;            # ditto
use 5.006_001;        # ditto; preferred for backwards compatibility
```


This is often useful if you need to check the current Perl version before using library modules that have changed in incompatible ways from older versions of Perl. (We try not to do this more than we have to.)

The `BEGIN` forces the `require` and `import` to happen at compile time. The `require` makes sure the module is loaded into memory if it hasn't been yet. The `import` is not a builtin—it's just an ordinary static method call into the `Module` package to tell the module to import the list of features back into the current package. The module can implement its `import` method any way it likes, though most modules just choose to derive their `import` method via inheritance from the `Exporter` class that is defined in the `Exporter` module. See [Exporter](#). If no `import` method can be found then the call is skipped.

If you do not want to call the package's `import` method (for instance, to stop your namespace from being altered), explicitly supply the empty list:

```
use Module ();
```

That is exactly equivalent to

```
BEGIN { require Module }
```

If the `VERSION` argument is present between `Module` and `LIST`, then the `use` will call the `VERSION` method in class `Module` with the given version as an argument. The default `VERSION` method, inherited from the `UNIVERSAL` class, croaks if the given version is larger than the value of the variable `$Module::VERSION`.

Again, there is a distinction between omitting `LIST` (`import` called with no arguments) and an explicit empty `LIST ()` (`import` not called). Note that there is no comma after `VERSION`!

Because this is a wide-open interface, pragmas (compiler directives) are also implemented this way. Currently implemented pragmas are:

```
use constant;
use diagnostics;
use integer;
use sigtrap    qw(SEGV BUS);
use strict     qw(subs vars refs);
use subs       qw(afunc blurfl);
use warnings   qw(all);
use sort       qw(stable _quicksort _mergesort);
```

Some of these pseudo-modules import semantics into the current block scope (like `strict` or `integer`, unlike ordinary modules, which import symbols into the current package (which are effective through the end of the file).

There's a corresponding `no` command that unimports meanings imported by `use`, i.e., it calls `unimport Module LIST` instead of `import`.

```
no integer;
no strict 'refs';
no warnings;
```

See [perlmodlib](#) for a list of standard modules and pragmas. See [perlrun](#) for the `-M` and `-m` command-line options to `perl` that give `use` functionality from the command-line.

utime LIST

Changes the access and modification times on each file of a list of files. The first two elements of the list must be the `NUMERICAL` access and modification times, in that order. Returns the number of files successfully changed. The inode change time of each file is set to the current time. This code has the same effect as the `touch` command if the files already exist:

```
#!/usr/bin/perl
$now = time;
utime $now, $now, @ARGV;
```

If the first two elements of the list are `undef`, then the `utime(2)` function in the C library will be called with a null second argument. On most systems, this will set the file's access and modification times to the current time. (i.e. equivalent to the example above.)

```
utime undef, undef, @ARGV;
```

values HASH

Returns a list consisting of all the values of the named hash. (In a scalar context, returns the number of values.) The values are returned in an apparently random order. The actual random order is subject to change in future versions of perl, but it is guaranteed to be the same order as either the `keys` or `each` function would produce on the same (unmodified) hash.

Note that the values are not copied, which means modifying them will modify the contents of the hash:

```
for (values %hash) { s/foo/bar/g } # modifies %hash values
for (@hash{keys %hash}) { s/foo/bar/g } # same
```

As a side effect, calling `values()` resets the HASH's internal iterator. See also `keys`, `each`, and `sort`.

vec EXPR,OFFSET,BITS

Treats the string in `EXPR` as a bit vector made up of elements of width `BITS`, and returns the value of the element specified by `OFFSET` as an unsigned integer. `BITS` therefore specifies the number of bits that are reserved for each element in the bit vector. This must be a power of two from 1 to 32 (or 64, if your platform supports that).

If `BITS` is 8, "elements" coincide with bytes of the input string.

If `BITS` is 16 or more, bytes of the input string are grouped into chunks of size `BITS/8`, and each group is converted to a number as with `pack()/unpack()` with big-endian formats `n/N` (and analogously for `BITS==64`). See *"pack"* for details.

If `bits` is 4 or less, the string is broken into bytes, then the bits of each byte are broken into `8/BITS` groups. Bits of a byte are numbered in a little-endian-ish way, as in `0x01`, `0x02`, `0x04`, `0x08`, `0x10`, `0x20`, `0x40`, `0x80`. For example, breaking the single input byte `chr(0x36)` into two groups gives a list `(0x6, 0x3)`; breaking it into 4 groups gives `(0x2, 0x1, 0x3, 0x0)`.

`vec` may also be assigned to, in which case parentheses are needed to give the expression the correct precedence as in

```
vec($image, $max_x * $x + $y, 8) = 3;
```

If the selected element is outside the string, the value 0 is returned. If an element off the end of the string is written to, Perl will first extend the string with sufficiently many zero bytes. It is an error to try to write off the beginning of the string (i.e. negative `OFFSET`).

The string should not contain any character with the value 255 (which can only happen if you're using UTF8 encoding). If it does, it will be treated as something which is not UTF8 encoded. When the `vec` was assigned to, other parts of your program will also no longer consider the string to be UTF8 encoded. In other words, if you do have such characters in your string, `vec()` will operate on the actual byte string, and not the conceptual character string.

Strings created with `vec` can also be manipulated with the logical operators `|`, `&`, `^`, and `~`. These operators will assume a bit vector operation is desired when both operands are strings. See *Bitwise String Operators in perlop*.

The following code will build up an ASCII string saying 'PerlPerlPerl'. The comments show the string after each step. Note that this code works in the same way on big-endian or little-endian machines.

```
my $foo = '';
vec($foo, 0, 32) = 0x5065726C;      # 'Perl'

# $foo eq "Perl" eq "\x50\x65\x72\x6C", 32 bits
print vec($foo, 0, 8);              # prints 80 == 0x50 == ord('P')

vec($foo, 2, 16) = 0x5065;          # 'PerlPe'
vec($foo, 3, 16) = 0x726C;          # 'PerlPerl'
vec($foo, 8, 8) = 0x50;              # 'PerlPerlP'
vec($foo, 9, 8) = 0x65;              # 'PerlPerlPe'
vec($foo, 20, 4) = 2;                # 'PerlPerlPe' . "\x02"
vec($foo, 21, 4) = 7;                # 'PerlPerlPer'
                                      # 'r' is "\x72"
vec($foo, 45, 2) = 3;                # 'PerlPerlPer' . "\x0c"
vec($foo, 93, 1) = 1;                # 'PerlPerlPer' . "\x2c"
vec($foo, 94, 1) = 1;                # 'PerlPerlPerl'
                                      # 'l' is "\x6c"
```

To transform a bit vector into a string or list of 0's and 1's, use these:

```
$bits = unpack("b*", $vector);
@bits = split("//", unpack("b*", $vector));
```

If you know the exact length in bits, it can be used in place of the *.

Here is an example to illustrate how the bits actually fall in place:

[illegible]

Regardless of the machine architecture on which it is run, the above example should print the

following table:

			0	1	2	3
		unpack("V",\$_)	01234567890123456789012345678901			

vec(\$_, 0, 1) = 1	==	1	10000000000000000000000000000000			
vec(\$_, 1, 1) = 1	==	2	01000000000000000000000000000000			
vec(\$_, 2, 1) = 1	==	4	00100000000000000000000000000000			
vec(\$_, 3, 1) = 1	==	8	00010000000000000000000000000000			
vec(\$_, 4, 1) = 1	==	16	00001000000000000000000000000000			
vec(\$_, 5, 1) = 1	==	32	00000100000000000000000000000000			
vec(\$_, 6, 1) = 1	==	64	00000010000000000000000000000000			
vec(\$_, 7, 1) = 1	==	128	00000001000000000000000000000000			
vec(\$_, 8, 1) = 1	==	256	00000000100000000000000000000000			
vec(\$_, 9, 1) = 1	==	512	00000000010000000000000000000000			
vec(\$_, 10, 1) = 1	==	1024	00000000001000000000000000000000			
vec(\$_, 11, 1) = 1	==	2048	00000000000100000000000000000000			
vec(\$_, 12, 1) = 1	==	4096	00000000000010000000000000000000			
vec(\$_, 13, 1) = 1	==	8192	00000000000001000000000000000000			
vec(\$_, 14, 1) = 1	==	16384	00000000000000100000000000000000			
vec(\$_, 15, 1) = 1	==	32768	00000000000000010000000000000000			
vec(\$_, 16, 1) = 1	==	65536	00000000000000001000000000000000			
vec(\$_, 17, 1) = 1	==	131072	00000000000000000100000000000000			
vec(\$_, 18, 1) = 1	==	262144	00000000000000000010000000000000			
vec(\$_, 19, 1) = 1	==	524288	00000000000000000001000000000000			
vec(\$_, 20, 1) = 1	==	1048576	00000000000000000000100000000000			
vec(\$_, 21, 1) = 1	==	2097152	00000000000000000000010000000000			
vec(\$_, 22, 1) = 1	==	4194304	00000000000000000000001000000000			
vec(\$_, 23, 1) = 1	==	8388608	00000000000000000000000100000000			
vec(\$_, 24, 1) = 1	==	16777216	00000000000000000000000010000000			
vec(\$_, 25, 1) = 1	==	33554432	00000000000000000000000001000000			
vec(\$_, 26, 1) = 1	==	67108864	00000000000000000000000000100000			
vec(\$_, 27, 1) = 1	==	134217728	00000000000000000000000000010000			
vec(\$_, 28, 1) = 1	==	268435456	00000000000000000000000000001000			
vec(\$_, 29, 1) = 1	==	536870912	00000000000000000000000000000100			
vec(\$_, 30, 1) = 1	==	1073741824	00000000000000000000000000000010			
vec(\$_, 31, 1) = 1	==	2147483648	00000000000000000000000000000001			
vec(\$_, 0, 2) = 1	==	1	10000000000000000000000000000000			
vec(\$_, 1, 2) = 1	==	4	00100000000000000000000000000000			
vec(\$_, 2, 2) = 1	==	16	00001000000000000000000000000000			
vec(\$_, 3, 2) = 1	==	64	00000010000000000000000000000000			
vec(\$_, 4, 2) = 1	==	256	00000000100000000000000000000000			
vec(\$_, 5, 2) = 1	==	1024	00000000001000000000000000000000			
vec(\$_, 6, 2) = 1	==	4096	00000000000010000000000000000000			
vec(\$_, 7, 2) = 1	==	16384	00000000000000100000000000000000			
vec(\$_, 8, 2) = 1	==	65536	00000000000000001000000000000000			
vec(\$_, 9, 2) = 1	==	262144	00000000000000000010000000000000			
vec(\$_, 10, 2) = 1	==	1048576	00000000000000000000100000000000			
vec(\$_, 11, 2) = 1	==	4194304	00000000000000000000001000000000			
vec(\$_, 12, 2) = 1	==	16777216	00000000000000000000000010000000			
vec(\$_, 13, 2) = 1	==	67108864	00000000000000000000000000100000			
vec(\$_, 14, 2) = 1	==	268435456	00000000000000000000000000001000			
vec(\$_, 15, 2) = 1	==	1073741824	00000000000000000000000000000010			
vec(\$_, 0, 2) = 2	==	2	01000000000000000000000000000000			
vec(\$_, 1, 2) = 2	==	8	00010000000000000000000000000000			

689

```

vec($_, 0, 8) = 4 == 4 00100000000000000000000000000000
vec($_, 1, 8) = 4 == 1024 00000000001000000000000000000000
vec($_, 2, 8) = 4 == 262144 000000000000000000001000000000000000
vec($_, 3, 8) = 4 == 67108864 0000000000000000000000000000100000
vec($_, 0, 8) = 8 == 8 0001000000000000000000000000000000
vec($_, 1, 8) = 8 == 2048 0000000000010000000000000000000000
vec($_, 2, 8) = 8 == 524288 000000000000000000000100000000000000
vec($_, 3, 8) = 8 == 134217728 0000000000000000000000000000100000
vec($_, 0, 8) = 16 == 16 0000100000000000000000000000000000
vec($_, 1, 8) = 16 == 4096 0000000000001000000000000000000000
vec($_, 2, 8) = 16 == 1048576 000000000000000000000100000000000000
vec($_, 3, 8) = 16 == 268435456 00000000000000000000000000001000
vec($_, 0, 8) = 32 == 32 0000010000000000000000000000000000
vec($_, 1, 8) = 32 == 8192 0000000000000100000000000000000000
vec($_, 2, 8) = 32 == 2097152 000000000000000000000001000000000000
vec($_, 3, 8) = 32 == 536870912 0000000000000000000000000000100
vec($_, 0, 8) = 64 == 64 0000001000000000000000000000000000
vec($_, 1, 8) = 64 == 16384 0000000000000010000000000000000000
vec($_, 2, 8) = 64 == 4194304 000000000000000000000001000000000000
vec($_, 3, 8) = 64 == 1073741824 000000000000000000000000000010
vec($_, 0, 8) = 128 == 128 0000000100000000000000000000000000
vec($_, 1, 8) = 128 == 32768 0000000000000000010000000000000000
vec($_, 2, 8) = 128 == 8388608 00000000000000000000000100000000
vec($_, 3, 8) = 128 == 2147483648 000000000000000000000000000001

```

wait Behaves like the `wait(2)` system call on your system: it waits for a child process to terminate and returns the pid of the deceased process, or `-1` if there are no child processes. The status is returned in `$?`. Note that a return value of `-1` could mean that child processes are being automatically reaped, as described in [perlipc](#).

waitpid PID,FLAGS

Waits for a particular child process to terminate and returns the pid of the deceased process, or `-1` if there is no such child process. On some systems, a value of `0` indicates that there are processes still running. The status is returned in `$?`. If you say

```

use POSIX ":sys_wait_h";
#...
do {
    $kid = waitpid(-1, WNOHANG);
} until $kid > 0;

```

then you can do a non-blocking wait for all pending zombie processes. Non-blocking wait is available on machines supporting either the `waitpid(2)` or `wait4(2)` system calls. However, waiting for a particular pid with `FLAGS` of `is` implemented everywhere. (Perl emulates the system call by remembering the status values of processes that have exited but have not been harvested by the Perl script yet.)

Note that on some systems, a return value of `-1` could mean that child processes are being automatically reaped. See [perlipc](#) for details, and for other examples.

wantarray

Returns true if the context of the currently executing subroutine is looking for a list value. Returns false if the context is looking for a scalar. Returns the undefined value if the context is looking for no value (void context).

```

return unless defined wantarray;    # don't bother doing more
my @a = complex_calculation();
return wantarray ? @a : "@a";

```

This function should have been named `wantlist()` instead.

warn LIST

Produces a message on `STDERR` just like `die`, but doesn't exit or throw an exception.

If `LIST` is empty and `$@` already contains a value (typically from a previous `eval`) that value is used after appending `"\t...caught"` to `$@`. This is useful for staying almost, but not entirely similar to `die`.

If `$@` is empty then the string `"Warning: Something's wrong"` is used.

No message is printed if there is a `$SIG{__WARN__}` handler installed. It is the handler's responsibility to deal with the message as it sees fit (like, for instance, converting it into a `die`). Most handlers must therefore make arrangements to actually display the warnings that they are not prepared to deal with, by calling `warn` again in the handler. Note that this is quite safe and will not produce an endless loop, since `__WARN__` hooks are not called from inside one.

You will find this behavior is slightly different from that of `$SIG{__DIE__}` handlers (which don't suppress the error text, but can instead call `die` again to change it).

Using a `__WARN__` handler provides a powerful way to silence all warnings (even the so-called mandatory ones). An example:

```
# wipe out *all* compile-time warnings
BEGIN { $SIG{'__WARN__'} = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20;           # no warning about duplicate my $foo,
                        # but hey, you asked for it!

# no compile-time or run-time warnings before here
$DOWARN = 1;

# run-time warnings enabled after here
warn "\$foo is alive and $foo!";    # does show up
```

See [perlvar](#) for details on setting `%SIG` entries, and for more examples. See the `Carp` module for other kinds of warnings using its `carp()` and `cluck()` functions.

write FILEHANDLE

write EXPR

write Writes a formatted record (possibly multi-line) to the specified `FILEHANDLE`, using the format associated with that file. By default the format for a file is the one having the same name as the filehandle, but the format for the current output channel (see the `select` function) may be set explicitly by assigning the name of the format to the `$~` variable.

Top of form processing is handled automatically: if there is insufficient room on the current page for the formatted record, the page is advanced by writing a form feed, a special top-of-page format is used to format the new page header, and then the record is written. By default the top-of-page format is the name of the filehandle with `"_TOP"` appended, but it may be dynamically set to the format of your choice by assigning the name to the `$^` variable while the filehandle is selected. The number of lines remaining on the current page is in variable `$-`, which can be set to force a new page.

If `FILEHANDLE` is unspecified, output goes to the current default output channel, which starts out as `STDOUT` but may be changed by the `select` operator. If the `FILEHANDLE` is an `EXPR`, then the expression is evaluated and the resulting string is used to look up the name of the `FILEHANDLE` at run time. For more on formats, see [perlform](#).

Note that `write` is *not* the opposite of `read`. Unfortunately.

`y///` The transliteration operator. Same as `tr///`. See [perlop](#).

NAME

perl guts – Introduction to the Perl API

DESCRIPTION

This document attempts to describe how to use the Perl API, as well as containing some info on the basic workings of the Perl core. It is far from complete and probably contains many errors. Please refer any questions or comments to the author below.

Variables**Datatypes**

Perl has three typedefs that handle Perl's three main data types:

```
SV   Scalar Value
AV   Array Value
HV   Hash Value
```

Each typedef has specific routines that manipulate the various data types.

What is an "IV"?

Perl uses a special typedef IV which is a simple signed integer type that is guaranteed to be large enough to hold a pointer (as well as an integer). Additionally, there is the UV, which is simply an unsigned IV.

Perl also uses two special typedefs, I32 and I16, which will always be at least 32-bits and 16-bits long, respectively. (Again, there are U32 and U16, as well.) They will usually be exactly 32 and 16 bits long, but on Crays they will both be 64 bits.

Working with SVs

An SV can be created and loaded with one command. There are five types of values that can be loaded: an integer value (IV), an unsigned integer value (UV), a double (NV), a string (PV), and another scalar (SV).

The seven routines are:

```
SV*  newSViv(IV);
SV*  newSVuv(UV);
SV*  newSVnv(double);
SV*  newSVpv(const char*, int);
SV*  newSVpvn(const char*, int);
SV*  newSVpvf(const char*, ...);
SV*  newSVsv(SV*);
```

If you require more complex initialisation you can create an empty SV with newSV(len). If len is 0 an empty SV of type NULL is returned, else an SV of type PV is returned with len + 1 (for the NUL) bytes of storage allocated, accessible via SvPVX. In both cases the SV has value undef.

```
SV*  newSV(0);    /* no storage allocated */
SV*  newSV(10);   /* 10 (+1) bytes of uninitialised storage allocated */
```

To change the value of an *already-existing* SV, there are eight routines:

```
void  sv_setiv(SV*, IV);
void  sv_setuv(SV*, UV);
void  sv_setnv(SV*, double);
void  sv_setpv(SV*, const char*);
void  sv_setpvn(SV*, const char*, int);
void  sv_setpvf(SV*, const char*, ...);
void  sv_vsetpvfn(SV*, const char*, STRLEN, va_list *, SV **, I32, bool *);
void  sv_setsv(SV*, SV*);
```

Notice that you can choose to specify the length of the string to be assigned by using sv_setpvn, newSVpvn, or newSVpv, or you may allow Perl to calculate the length by using sv_setpv or by

specifying 0 as the second argument to `newSVpv`. Be warned, though, that Perl will determine the string's length by using `strlen`, which depends on the string terminating with a NUL character.

The arguments of `sv_setpvf` are processed like `sprintf`, and the formatted output becomes the value.

`sv_vsetpvfn` is an analogue of `vsprintf`, but it allows you to specify either a pointer to a variable argument list or the address and length of an array of SVs. The last argument points to a boolean; on return, if that boolean is true, then locale-specific information has been used to format the string, and the string's contents are therefore untrustworthy (see [perlsec](#)). This pointer may be NULL if that information is not important. Note that this function requires you to specify the length of the format.

STRLEN is an integer type (`Size_t`, usually defined as `size_t` in `config.h`) guaranteed to be large enough to represent the size of any string that perl can handle.

The `sv_set*`() functions are not generic enough to operate on values that have "magic". See [Magic Virtual Tables](#) later in this document.

All SVs that contain strings should be terminated with a NUL character. If it is not NUL-terminated there is a risk of core dumps and corruptions from code which passes the string to C functions or system calls which expect a NUL-terminated string. Perl's own functions typically add a trailing NUL for this reason. Nevertheless, you should be very careful when you pass a string stored in an SV to a C function or system call.

To access the actual value that an SV points to, you can use the macros:

```
SvIV(SV*)
SvUV(SV*)
SvNV(SV*)
SvPV(SV*, STRLEN len)
SvPV_nolen(SV*)
```

which will automatically coerce the actual scalar type into an IV, UV, double, or string.

In the `SvPV` macro, the length of the string returned is placed into the variable `len` (this is a macro, so you do *not* use `&len`). If you do not care what the length of the data is, use the `SvPV_nolen` macro. Historically the `SvPV` macro with the global variable `PL_na` has been used in this case. But that can be quite inefficient because `PL_na` must be accessed in thread-local storage in threaded Perl. In any case, remember that Perl allows arbitrary strings of data that may both contain NULs and might not be terminated by a NUL.

Also remember that C doesn't allow you to safely say `foo(SvPV(s, len), len)`; It might work with your compiler, but it won't work for everyone. Break this sort of statement up into separate assignments:

```
SV *s;
STRLEN len;
char * ptr;
ptr = SvPV(s, len);
foo(ptr, len);
```

If you want to know if the scalar value is TRUE, you can use:

```
SvTRUE(SV*)
```

Although Perl will automatically grow strings for you, if you need to force Perl to allocate more memory for your SV, you can use the macro

```
SvGROW(SV*, STRLEN newlen)
```

which will determine if more memory needs to be allocated. If so, it will call the function `sv_grow`. Note that `SvGROW` can only increase, not decrease, the allocated memory of an SV and that it does not automatically add a byte for the a trailing NUL (perl's own string functions typically do `SvGROW(sv, len + 1)`).

If you have an SV and want to know what kind of data Perl thinks is stored in it, you can use the following macros to check the type of SV you have.

```
SvIOK(SV*)
SvNOK(SV*)
SvPOK(SV*)
```

You can get and set the current length of the string stored in an SV with the following macros:

```
SvCUR(SV*)
SvCUR_set(SV*, I32 val)
```

You can also get a pointer to the end of the string stored in the SV with the macro:

```
SvEND(SV*)
```

But note that these last three macros are valid only if SvPOK() is true.

If you want to append something to the end of string stored in an SV*, you can use the following functions:

```
void sv_catpv(SV*, const char*);
void sv_catpvN(SV*, const char*, STRLEN);
void sv_catpvf(SV*, const char*, ...);
void sv_vcatpvfn(SV*, const char*, STRLEN, va_list *, SV **, I32, bool);
void sv_catsv(SV*, SV*);
```

The first function calculates the length of the string to be appended by using strlen. In the second, you specify the length of the string yourself. The third function processes its arguments like sprintf and appends the formatted output. The fourth function works like vsprintf. You can specify the address and length of an array of SVs instead of the va_list argument. The fifth function extends the string stored in the first SV with the string stored in the second SV. It also forces the second SV to be interpreted as a string.

The sv_cat*() functions are not generic enough to operate on values that have "magic". See [Magic Virtual Tables](#) later in this document.

If you know the name of a scalar variable, you can get a pointer to its SV by using the following:

```
SV* get_sv("package::varname", FALSE);
```

This returns NULL if the variable does not exist.

If you want to know if this variable (or any other SV) is actually defined, you can call:

```
SvOK(SV*)
```

The scalar undef value is stored in an SV instance called PL_sv_undef. Its address can be used whenever an SV* is needed.

There are also the two values PL_sv_yes and PL_sv_no, which contain Boolean TRUE and FALSE values, respectively. Like PL_sv_undef, their addresses can be used whenever an SV* is needed.

Do not be fooled into thinking that (SV *) 0 is the same as &PL_sv_undef. Take this code:

```
SV* sv = (SV*) 0;
if (I-am-to-return-a-real-value) {
    sv = sv_2mortal(newSViv(42));
}
sv_setsv(ST(0), sv);
```

This code tries to return a new SV (which contains the value 42) if it should return a real value, or undef otherwise. Instead it has returned a NULL pointer which, somewhere down the line, will cause a segmentation violation, bus error, or just weird results. Change the zero to &PL_sv_undef in the first line and all will be well.

To free an SV that you've created, call `SvREFCNT_dec(SV*)`. Normally this call is not necessary (see [Reference Counts and Mortality](#)).

Offsets

Perl provides the function `sv_chop` to efficiently remove characters from the beginning of a string; you give it an SV and a pointer to somewhere inside the PV, and it discards everything before the pointer. The efficiency comes by means of a little hack: instead of actually removing the characters, `sv_chop` sets the flag `OOK` (offset OK) to signal to other functions that the offset hack is in effect, and it puts the number of bytes chopped off into the IV field of the SV. It then moves the PV pointer (called `SvPVX`) forward that many bytes, and adjusts `SvCUR` and `SvLEN`.

Hence, at this point, the start of the buffer that we allocated lives at `SvPVX(sv) - SvIV(sv)` in memory and the PV pointer is pointing into the middle of this allocated storage.

This is best demonstrated by example:

```
% ./perl -Ilib -MDevel::Peek -le '$a="12345"; $a=~s/./;/; Dump($a)'
SV = PVIV(0x8128450) at 0x81340f0
  REFCNT = 1
  FLAGS = (POK,OOK,pPOK)
  IV = 1   (OFFSET)
  PV = 0x8135781 ( "1" . ) "2345"\0
  CUR = 4
  LEN = 5
```

Here the number of bytes chopped off (1) is put into IV, and `Devel::Peek::Dump` helpfully reminds us that this is an offset. The portion of the string between the "real" and the "fake" beginnings is shown in parentheses, and the values of `SvCUR` and `SvLEN` reflect the fake beginning, not the real one.

Something similar to the offset hack is performed on AVs to enable efficient shifting and splicing off the beginning of the array; while `AvARRAY` points to the first element in the array that is visible from Perl, `AvALLOC` points to the real start of the C array. These are usually the same, but a `shift` operation can be carried out by increasing `AvARRAY` by one and decreasing `AvFILL` and `AvLEN`. Again, the location of the real start of the C array only comes into play when freeing the array. See `av_shift` in *av.c*.

What's Really Stored in an SV?

Recall that the usual method of determining the type of scalar you have is to use `Sv*OK` macros. Because a scalar can be both a number and a string, usually these macros will always return TRUE and calling the `Sv*V` macros will do the appropriate conversion of string to integer/double or integer/double to string.

If you *really* need to know if you have an integer, double, or string pointer in an SV, you can use the following three macros instead:

```
SvIOKp(SV*)
SvNOKp(SV*)
SvPOKp(SV*)
```

These will tell you if you truly have an integer, double, or string pointer stored in your SV. The "p" stands for private.

There are various ways in which the private and public flags may differ. For example, a tied SV may have a valid underlying value in the IV slot (so `SvIOKp` is true), but the data should be accessed via the `FETCH` routine rather than directly, so `SvIOK` is false. Another is when numeric conversion has occurred and precision has been lost: only the private flag is set on 'lossy' values. So when an NV is converted to an IV with loss, `SvIOKp`, `SvNOKp` and `SvNOK` will be set, while `SvIOK` won't be.

In general, though, it's best to use the `Sv*V` macros.

Working with AVs

There are two ways to create and load an AV. The first method creates an empty AV:

```
AV* newAV();
```

The second method both creates the AV and initially populates it with SVs:

```
AV* av_make(I32 num, SV **ptr);
```

The second argument points to an array containing num SV*'s. Once the AV has been created, the SVs can be destroyed, if so desired.

Once the AV has been created, the following operations are possible on AVs:

```
void av_push(AV*, SV*);
SV* av_pop(AV*);
SV* av_shift(AV*);
void av_unshift(AV*, I32 num);
```

These should be familiar operations, with the exception of `av_unshift`. This routine adds num elements at the front of the array with the undef value. You must then use `av_store` (described below) to assign values to these new elements.

Here are some other functions:

```
I32 av_len(AV*);
SV** av_fetch(AV*, I32 key, I32 lval);
SV** av_store(AV*, I32 key, SV* val);
```

The `av_len` function returns the highest index value in array (just like `$#array` in Perl). If the array is empty, -1 is returned. The `av_fetch` function returns the value at index key, but if lval is non-zero, then `av_fetch` will store an undef value at that index. The `av_store` function stores the value val at index key, and does not increment the reference count of val. Thus the caller is responsible for taking care of that, and if `av_store` returns NULL, the caller will have to decrement the reference count to avoid a memory leak. Note that `av_fetch` and `av_store` both return SV**'s, not SV*'s as their return value.

```
void av_clear(AV*);
void av_undef(AV*);
void av_extend(AV*, I32 key);
```

The `av_clear` function deletes all the elements in the AV* array, but does not actually delete the array itself. The `av_undef` function will delete all the elements in the array plus the array itself. The `av_extend` function extends the array so that it contains at least key+1 elements. If key+1 is less than the currently allocated length of the array, then nothing is done.

If you know the name of an array variable, you can get a pointer to its AV by using the following:

```
AV* get_av("package::varname", FALSE);
```

This returns NULL if the variable does not exist.

See [Understanding the Magic of Tied Hashes and Arrays](#) for more information on how to use the array access functions on tied arrays.

Working with HVs

To create an HV, you use the following routine:

```
HV* newHV();
```

Once the HV has been created, the following operations are possible on HVs:

```
SV** hv_store(HV*, const char* key, U32 klen, SV* val, U32 hash);
SV** hv_fetch(HV*, const char* key, U32 klen, I32 lval);
```

The `klen` parameter is the length of the key being passed in (Note that you cannot pass 0 in as a value of `klen` to tell Perl to measure the length of the key). The `val` argument contains the SV pointer to the scalar being stored, and `hash` is the precomputed hash value (zero if you want `hv_store` to calculate it for you). The `lval` parameter indicates whether this fetch is actually a part of a store operation, in which case a new undefined value will be added to the HV with the supplied key and `hv_fetch` will return as if the value had already existed.

Remember that `hv_store` and `hv_fetch` return `SV**`'s and not just `SV*`. To access the scalar value, you must first dereference the return value. However, you should check to make sure that the return value is not NULL before dereferencing it.

These two functions check if a hash table entry exists, and deletes it.

```
bool  hv_exists(HV*, const char* key, U32 klen);
SV*   hv_delete(HV*, const char* key, U32 klen, I32 flags);
```

If `flags` does not include the `G_DISCARD` flag then `hv_delete` will create and return a mortal copy of the deleted value.

And more miscellaneous functions:

```
void   hv_clear(HV*);
void   hv_undef(HV*);
```

Like their AV counterparts, `hv_clear` deletes all the entries in the hash table but does not actually delete the hash table. The `hv_undef` deletes both the entries and the hash table itself.

Perl keeps the actual data in linked list of structures with a typedef of HE. These contain the actual key and value pointers (plus extra administrative overhead). The key is a string pointer; the value is an `SV*`. However, once you have an `HE*`, to get the actual key and value, use the routines specified below.

```
I32    hv_iterinit(HV*);
        /* Prepares starting point to traverse hash table */
HE*    hv_iternext(HV*);
        /* Get the next entry, and return a pointer to a
           structure that has both the key and value */
char*   hv_iterkey(HE* entry, I32* retlen);
        /* Get the key from an HE structure and also return
           the length of the key string */
SV*     hv_ival(HV*, HE* entry);
        /* Return an SV pointer to the value of the HE
           structure */
SV*     hv_iternextsv(HV*, char** key, I32* retlen);
        /* This convenience routine combines hv_iternext,
           hv_iterkey, and hv_ival. The key and retlen
           arguments are return values for the key and its
           length. The value is returned in the SV* argument */
```

If you know the name of a hash variable, you can get a pointer to its HV by using the following:

```
HV*    get_hv("package::varname", FALSE);
```

This returns NULL if the variable does not exist.

The hash algorithm is defined in the `PERL_HASH(hash, key, klen)` macro:

```
hash = 0;
while (klen--)
    hash = (hash * 33) + *key++;
hash = hash + (hash >> 5);          /* after 5.6 */
```

The last step was added in version 5.6 to improve distribution of lower bits in the resulting hash value.

See [Understanding the Magic of Tied Hashes and Arrays](#) for more information on how to use the hash access functions on tied hashes.

Hash API Extensions

Beginning with version 5.004, the following functions are also supported:

```
HE*      hv_fetch_ent  (HV* tb, SV* key, I32 lval, U32 hash);
HE*      hv_store_ent  (HV* tb, SV* key, SV* val, U32 hash);

bool     hv_exists_ent (HV* tb, SV* key, U32 hash);
SV*      hv_delete_ent (HV* tb, SV* key, I32 flags, U32 hash);
SV*      hv_iterkeysv  (HE* entry);
```

Note that these functions take SV* keys, which simplifies writing of extension code that deals with hash structures. These functions also allow passing of SV* keys to tie functions without forcing you to stringify the keys (unlike the previous set of functions).

They also return and accept whole hash entries (HE*), making their use more efficient (since the hash number for a particular string doesn't have to be recomputed every time). See [perlapi](#) for detailed descriptions.

The following macros must always be used to access the contents of hash entries. Note that the arguments to these macros must be simple variables, since they may get evaluated more than once. See [perlapi](#) for detailed descriptions of these macros.

```
HePV(HE* he, STRLEN len)
HeVAL(HE* he)
HeHASH(HE* he)
HeSVKEY(HE* he)
HeSVKEY_force(HE* he)
HeSVKEY_set(HE* he, SV* sv)
```

These two lower level macros are defined, but must only be used when dealing with keys that are not SV*s:

```
HeKEY(HE* he)
HeKLEN(HE* he)
```

Note that both hv_store and hv_store_ent do not increment the reference count of the stored val, which is the caller's responsibility. If these functions return a NULL value, the caller will usually have to decrement the reference count of val to avoid a memory leak.

References

References are a special type of scalar that point to other data types (including references).

To create a reference, use either of the following functions:

```
SV* newRV_inc((SV*) thing);
SV* newRV_noinc((SV*) thing);
```

The thing argument can be any of an SV*, AV*, or HV*. The functions are identical except that newRV_inc increments the reference count of the thing, while newRV_noinc does not. For historical reasons, newRV is a synonym for newRV_inc.

Once you have a reference, you can use the following macro to dereference the reference:

```
SvRV(SV*)
```

then call the appropriate routines, casting the returned SV* to either an AV* or HV*, if required.

To determine if an SV is a reference, you can use the following macro:

```
SvROK(SV*)
```

To discover what type of value the reference refers to, use the following macro and then check the return value.

```
SvTYPE(SvRV(SV*))
```

The most useful types that will be returned are:

```
SVt_IV      Scalar
SVt_NV      Scalar
SVt_PV      Scalar
SVt_RV      Scalar
SVt_PVAV    Array
SVt_PVHV    Hash
SVt_PVCV    Code
SVt_PGV     Glob (possible a file handle)
SVt_PVMG    Blessed or Magical Scalar
```

See the `sv.h` header file for more details.

Blessed References and Class Objects

References are also used to support object-oriented programming. In the OO lexicon, an object is simply a reference that has been blessed into a package (or class). Once blessed, the programmer may now use the reference to access the various methods in the class.

A reference can be blessed into a package with the following function:

```
SV* sv_bless(SV* sv, HV* stash);
```

The `sv` argument must be a reference. The `stash` argument specifies which class the reference will belong to. See [Stashes and Globs](#) for information on converting class names into stashes.

/ Still under construction */*

Upgrades `rv` to reference if not already one. Creates new `SV` for `rv` to point to. If `classname` is non-null, the `SV` is blessed into the specified class. `SV` is returned.

```
SV* newSVrv(SV* rv, const char* classname);
```

Copies integer, unsigned integer or double into an `SV` whose reference is `rv`. `SV` is blessed if `classname` is non-null.

```
SV* sv_setref_iv(SV* rv, const char* classname, IV iv);
SV* sv_setref_uv(SV* rv, const char* classname, UV uv);
SV* sv_setref_nv(SV* rv, const char* classname, NV iv);
```

Copies the pointer value (*the address, not the string!*) into an `SV` whose reference is `rv`. `SV` is blessed if `classname` is non-null.

```
SV* sv_setref_pv(SV* rv, const char* classname, PV iv);
```

Copies string into an `SV` whose reference is `rv`. Set `length` to 0 to let Perl calculate the string length. `SV` is blessed if `classname` is non-null.

```
SV* sv_setref_pvn(SV* rv, const char* classname, PV iv, STRLEN length);
```

Tests whether the `SV` is blessed into the specified class. It does not check inheritance relationships.

```
int sv_isa(SV* sv, const char* name);
```

Tests whether the `SV` is a reference to a blessed object.

```
int sv_isobject(SV* sv);
```


Tests whether the SV is derived from the specified class. SV can be either a reference to a blessed object or a string containing a class name. This is the function implementing the `UNIVERSAL::isa` functionality.

```
bool sv_derived_from(SV* sv, const char* name);
```

To check if you've got an object derived from a specific class you have to write:

```
if (sv_isobject(sv) && sv_derived_from(sv, class)) { ... }
```

Creating New Variables

To create a new Perl variable with an undef value which can be accessed from your Perl script, use the following routines, depending on the variable type.

```
SV*  get_sv("package::varname", TRUE);
AV*  get_av("package::varname", TRUE);
HV*  get_hv("package::varname", TRUE);
```

Notice the use of TRUE as the second parameter. The new variable can now be set, using the routines appropriate to the data type.

There are additional macros whose values may be bitwise OR'ed with the TRUE argument to enable certain extra features. Those bits are:

GV_ADDMULTI

Marks the variable as multiply defined, thus preventing the:

```
Name <varname> used only once: possible typo
warning.
```

GV_ADDWARN

Issues the warning:

```
Had to create <varname> unexpectedly
if the variable did not exist before the function was called.
```

If you do not specify a package name, the variable is created in the current package.

Reference Counts and Mortality

Perl uses a reference count-driven garbage collection mechanism. SVs, AVs, or HVs (xV for short in the following) start their life with a reference count of 1. If the reference count of an xV ever drops to 0, then it will be destroyed and its memory made available for reuse.

This normally doesn't happen at the Perl level unless a variable is undef'ed or the last variable holding a reference to it is changed or overwritten. At the internal level, however, reference counts can be manipulated with the following macros:

```
int SvREFCNT(SV* sv);
SV* SvREFCNT_inc(SV* sv);
void SvREFCNT_dec(SV* sv);
```

However, there is one other function which manipulates the reference count of its argument. The `newRV_inc` function, you will recall, creates a reference to the specified argument. As a side effect, it increments the argument's reference count. If this is not what you want, use `newRV_noinc` instead.

For example, imagine you want to return a reference from an XSUB function. Inside the XSUB routine, you create an SV which initially has a reference count of one. Then you call `newRV_inc`, passing it the just-created SV. This returns the reference as a new SV, but the reference count of the SV you passed to `newRV_inc` has been incremented to two. Now you return the reference from the XSUB routine and forget about the SV. But Perl hasn't! Whenever the returned reference is destroyed, the reference count of the original SV is decreased to one and nothing happens. The SV will hang around without any way to access it until Perl itself terminates. This is a memory leak.

The correct procedure, then, is to use `newRV_noinc` instead of `newRV_inc`. Then, if and when the last reference is destroyed, the reference count of the SV will go to zero and it will be destroyed, stopping any memory leak.

There are some convenience functions available that can help with the destruction of xVs. These functions introduce the concept of "mortality". An xV that is mortal has had its reference count marked to be decremented, but not actually decremented, until "a short time later". Generally the term "short time later" means a single Perl statement, such as a call to an XSUB function. The actual determinant for when mortal xVs have their reference count decremented depends on two macros, `SAVETMPS` and `FREETMPS`. See [perlcalls](#) and [perlxs](#) for more details on these macros.

"Mortalization" then is at its simplest a deferred `SvREFCNT_dec`. However, if you mortalize a variable twice, the reference count will later be decremented twice.

"Mortal" SVs are mainly used for SVs that are placed on perl's stack. For example an SV which is created just to pass a number to a called sub is made mortal to have it cleaned up automatically when stack is popped. Similarly results returned by XSUBs (which go in the stack) are often made mortal.

To create a mortal variable, use the functions:

```
SV*  sv_newmortal()
SV*  sv_2mortal(SV*)
SV*  sv_mortalcopy(SV*)
```

The first call creates a mortal SV (with no value), the second converts an existing SV to a mortal SV (and thus defers a call to `SvREFCNT_dec`), and the third creates a mortal copy of an existing SV. Because `sv_newmortal` gives the new SV no value, it must normally be given one via `sv_setpv`, `sv_setiv`, etc.:

```
SV *tmp = sv_newmortal();
sv_setiv(tmp, an_integer);
```

As that is multiple C statements it is quite common so see this idiom instead:

```
SV *tmp = sv_2mortal(newSViv(an_integer));
```

You should be careful about creating mortal variables. Strange things can happen if you make the same value mortal within multiple contexts, or if you make a variable mortal multiple times. Thinking of "Mortalization" as deferred `SvREFCNT_dec` should help to minimize such problems. For example if you are passing an SV which you *know* has high enough `REFCNT` to survive its use on the stack you need not do any mortalization. If you are not sure then doing an `SvREFCNT_inc` and `sv_2mortal`, or making a `sv_mortalcopy` is safer.

The mortal routines are not just for SVs — AVs and HVs can be made mortal by passing their address (type-casted to `SV*`) to the `sv_2mortal` or `sv_mortalcopy` routines.

Stashes and Globs

A "stash" is a hash that contains all of the different objects that are contained within a package. Each key of the stash is a symbol name (shared by all the different types of objects that have the same name), and each value in the hash table is a GV (Glob Value). This GV in turn contains references to the various objects of that name, including (but not limited to) the following:

```
Scalar Value
Array Value
Hash Value
I/O Handle
Format
Subroutine
```

There is a single stash called "PL_defstash" that holds the items that exist in the "main" package. To get at the items in other packages, append the string "::" to the package name. The items in the "Foo" package are

in the stash "Foo::" in PL_defstash. The items in the "Bar::Baz" package are in the stash "Baz::" in "Bar::"'s stash.

To get the stash pointer for a particular package, use the function:

```
HV*  gv_stashpv(const char* name, I32 create)
HV*  gv_stashsv(SV*, I32 create)
```

The first function takes a literal string, the second uses the string stored in the SV. Remember that a stash is just a hash table, so you get back an HV*. The create flag will create a new package if it is set.

The name that gv_stash*v wants is the name of the package whose symbol table you want. The default package is called main. If you have multiply nested packages, pass their names to gv_stash*v, separated by :: as in the Perl language itself.

Alternately, if you have an SV that is a blessed reference, you can find out the stash pointer by using:

```
HV*  SvSTASH(SvRV(SV*)) ;
```

then use the following to get the package name itself:

```
char*  HvNAME(HV* stash) ;
```

If you need to bless or re-bless an object you can use the following function:

```
SV*  sv_bless(SV*, HV* stash)
```

where the first argument, an SV*, must be a reference, and the second argument is a stash. The returned SV* can now be used in the same way as any other SV.

For more information on references and blessings, consult [perlref](#).

Double-Typed SVs

Scalar variables normally contain only one type of value, an integer, double, pointer, or reference. Perl will automatically convert the actual scalar data from the stored type into the requested type.

Some scalar variables contain more than one type of scalar data. For example, the variable \$! contains either the numeric value of errno or its string equivalent from either strerror or sys_errlist[].

To force multiple data values into an SV, you must do two things: use the sv_set*v routines to add the additional scalar type, then set a flag so that Perl will believe it contains more than one type of data. The four macros to set the flags are:

```
SvIOK_on
SvNOK_on
SvPOK_on
SvROK_on
```

The particular macro you must use depends on which sv_set*v routine you called first. This is because every sv_set*v routine turns on only the bit for the particular type of data being set, and turns off all the rest.

For example, to create a new Perl variable called "dberror" that contains both the numeric and descriptive string error values, you could use the following code:

```
extern int  dberror;
extern char *dberror_list;

SV* sv = get_sv("dberror", TRUE);
sv_setiv(sv, (IV) dberror);
sv_setpv(sv, dberror_list[dberror]);
SvIOK_on(sv);
```

If the order of sv_setiv and sv_setpv had been reversed, then the macro SvPOK_on would need to be called instead of SvIOK_on.

Magic Variables

[This section still under construction. Ignore everything here. Post no bills. Everything not permitted is forbidden.]

Any SV may be magical, that is, it has special features that a normal SV does not have. These features are stored in the SV structure in a linked list of `struct magic`'s, typedef'ed to `MAGIC`.

```
struct magic {
    MAGIC*      mg_moremagic;
    MGVTBL*     mg_virtual;
    U16         mg_private;
    char        mg_type;
    U8          mg_flags;
    SV*         mg_obj;
    char*       mg_ptr;
    I32         mg_len;
};
```

Note this is current as of patchlevel 0, and could change at any time.

Assigning Magic

Perl adds magic to an SV using the `sv_magic` function:

```
void sv_magic(SV* sv, SV* obj, int how, const char* name, I32 namlen);
```

The `sv` argument is a pointer to the SV that is to acquire a new magical feature.

If `sv` is not already magical, Perl uses the `SvUPGRADE` macro to convert `sv` to type `SVt_PVMG`. Perl then continues by adding new magic to the beginning of the linked list of magical features. Any prior entry of the same type of magic is deleted. Note that this can be overridden, and multiple instances of the same type of magic can be associated with an SV.

The `name` and `namlen` arguments are used to associate a string with the magic, typically the name of a variable. `namlen` is stored in the `mg_len` field and if `name` is non-null and `namlen >= 0` a malloc'd copy of the name is stored in `mg_ptr` field.

The `sv_magic` function uses `how` to determine which, if any, predefined "Magic Virtual Table" should be assigned to the `mg_virtual` field. See the "Magic Virtual Table" section below. The `how` argument is also stored in the `mg_type` field. The value of `how` should be chosen from the set of macros `PERL_MAGIC_foo` found `perl.h`. Note that before these macros were added, Perl internals used to directly use character literals, so you may occasionally come across old code or documentation referring to 'U' magic rather than `PERL_MAGIC_uvar` for example.

The `obj` argument is stored in the `mg_obj` field of the `MAGIC` structure. If it is not the same as the `sv` argument, the reference count of the `obj` object is incremented. If it is the same, or if the `how` argument is `PERL_MAGIC_arylen`, or if it is a `NULL` pointer, then `obj` is merely stored, without the reference count being incremented.

There is also a function to add magic to an HV:

```
void hv_magic(HV *hv, GV *gv, int how);
```

This simply calls `sv_magic` and coerces the `gv` argument into an SV.

To remove the magic from an SV, call the function `sv_unmagic`:

```
void sv_unmagic(SV *sv, int type);
```

The `type` argument should be equal to the `how` value when the SV was initially made magical.

Magic Virtual Tables

The `mg_virtual` field in the `MAGIC` structure is a pointer to an `MGVTBL`, which is a structure of function pointers and stands for "Magic Virtual Table" to handle the various operations that might be applied to that variable.

The `MGVTBL` has five pointers to the following routine types:

```
int  (*svt_get)(SV* sv, MAGIC* mg);
int  (*svt_set)(SV* sv, MAGIC* mg);
U32  (*svt_len)(SV* sv, MAGIC* mg);
int  (*svt_clear)(SV* sv, MAGIC* mg);
int  (*svt_free)(SV* sv, MAGIC* mg);
```

This `MGVTBL` structure is set at compile-time in `perl.h` and there are currently 19 types (or 21 with overloading turned on). These different structures contain pointers to various routines that perform additional actions depending on which function is being called.

Function pointer	Action taken
-----	-----
<code>svt_get</code>	Do something before the value of the SV is retrieved.
<code>svt_set</code>	Do something after the SV is assigned a value.
<code>svt_len</code>	Report on the SV's length.
<code>svt_clear</code>	Clear something the SV represents.
<code>svt_free</code>	Free any extra storage associated with the SV.

For instance, the `MGVTBL` structure called `vtbl_sv` (which corresponds to an `mg_type` of `PERL_MAGIC_sv`) contains:

```
{ magic_get, magic_set, magic_len, 0, 0 }
```

Thus, when an SV is determined to be magical and of type `PERL_MAGIC_sv`, if a get operation is being performed, the routine `magic_get` is called. All the various routines for the various magical types begin with `magic_`. NOTE: the magic routines are not considered part of the Perl API, and may not be exported by the Perl library.

The current kinds of Magic Virtual Tables are:

mg_type (old-style char and macro)	MGVTBL	Type of magic
-----	-----	-----
\0 PERL_MAGIC_sv	vtbl_sv	Special scalar variable
A PERL_MAGIC_overload	vtbl_amagic	%OVERLOAD hash
a PERL_MAGIC_overload_elem	vtbl_amagic_elem	%OVERLOAD hash element
c PERL_MAGIC_overload_table	(none)	Holds overload table (AMT) on stash
B PERL_MAGIC_bm	vtbl_bm	Boyer-Moore (fast string search)
D PERL_MAGIC_regdata	vtbl_regdata	Regex match position data (@+ and @- vars)
d PERL_MAGIC_regdatum	vtbl_regdatum	Regex match position data element
E PERL_MAGIC_env	vtbl_env	%ENV hash
e PERL_MAGIC_envelem	vtbl_envelem	%ENV hash element
f PERL_MAGIC_fm	vtbl_fm	Formline ('compiled' format)
g PERL_MAGIC_regex_global	vtbl_mglob	m//g target / study()ed string
I PERL_MAGIC_isa	vtbl_isa	@ISA array
i PERL_MAGIC_isaelem	vtbl_isaelem	@ISA array element
k PERL_MAGIC_nkeys	vtbl_nkeys	scalar(keys()) lvalue
L PERL_MAGIC_dbfile	(none)	Debugger %_<filename

l	PERL_MAGIC_dbline	vtbl_dbline	Debugger %_<filename element
m	PERL_MAGIC_mutex	vtbl_mutex	???
o	PERL_MAGIC_collxfrm	vtbl_collxfrm	Locale collate transformation
P	PERL_MAGIC_tied	vtbl_pack	Tied array or hash
p	PERL_MAGIC_tiedelem	vtbl_packelem	Tied array or hash element
q	PERL_MAGIC_tiedscalar	vtbl_packelem	Tied scalar or handle
r	PERL_MAGIC_qr	vtbl_qr	precompiled qr// regex
S	PERL_MAGIC_sig	vtbl_sig	%SIG hash
s	PERL_MAGIC_sigelem	vtbl_sigelem	%SIG hash element
t	PERL_MAGIC_taint	vtbl_taint	Taintedness
U	PERL_MAGIC_uvar	vtbl_uvar	Available for use by extensions
v	PERL_MAGIC_vec	vtbl_vec	vec() lvalue
x	PERL_MAGIC_substr	vtbl_substr	substr() lvalue
y	PERL_MAGIC_defelem	vtbl_defelem	Shadow "foreach" iterator variable / smart parameter vivification
*	PERL_MAGIC_glob	vtbl_glob	GV (typeglob)
#	PERL_MAGIC_arylen	vtbl_arylen	Array length (\$#ary)
.	PERL_MAGIC_pos	vtbl_pos	pos() lvalue
<	PERL_MAGIC_backref	vtbl_backref	???
~	PERL_MAGIC_ext	(none)	Available for use by extensions

When an uppercase and lowercase letter both exist in the table, then the uppercase letter is used to represent some kind of composite type (a list or a hash), and the lowercase letter is used to represent an element of that composite type. Some internals code makes use of this case relationship.

The PERL_MAGIC_ext and PERL_MAGIC_uvar magic types are defined specifically for use by extensions and will not be used by perl itself. Extensions can use PERL_MAGIC_ext magic to ‘attach’ private information to variables (typically objects). This is especially useful because there is no way for normal perl code to corrupt this private information (unlike using extra elements of a hash object).

Similarly, PERL_MAGIC_uvar magic can be used much like tie() to call a C function any time a scalar’s value is used or changed. The MAGIC’s mg_ptr field points to a ufuncs structure:

```
struct ufuncs {
    I32 (*uf_val) (pTHX_ IV, SV*);
    I32 (*uf_set) (pTHX_ IV, SV*);
    IV uf_index;
};
```

When the SV is read from or written to, the uf_val or uf_set function will be called with uf_index as the first arg and a pointer to the SV as the second. A simple example of how to add PERL_MAGIC_uvar magic is shown below. Note that the ufuncs structure is copied by sv_magic, so you can safely allocate it on the stack.

```
void
Umagic(sv)
    SV *sv;
PREINIT:
    struct ufuncs uf;
CODE:
    uf.uf_val = &my_get_fn;
    uf.uf_set = &my_set_fn;
    uf.uf_index = 0;
    sv_magic(sv, 0, PERL_MAGIC_uvar, (char*)&uf, sizeof(uf));
```

Note that because multiple extensions may be using PERL_MAGIC_ext or PERL_MAGIC_uvar magic, it is important for extensions to take extra care to avoid conflict. Typically only using the magic on objects

blessed into the same class as the extension is sufficient. For `PERL_MAGIC_ext` magic, it may also be appropriate to add an I32 ‘signature’ at the top of the private data area and check that.

Also note that the `sv_set*()` and `sv_cat*()` functions described earlier do **not** invoke ‘set’ magic on their targets. This must be done by the user either by calling the `SvSETMAGIC()` macro after calling these functions, or by using one of the `sv_set*_mg()` or `sv_cat*_mg()` functions. Similarly, generic C code must call the `SvGETMAGIC()` macro to invoke any ‘get’ magic if they use an SV obtained from external sources in functions that don’t handle magic. See [perlapi](#) for a description of these functions. For example, calls to the `sv_cat*()` functions typically need to be followed by `SvSETMAGIC()`, but they don’t need a prior `SvGETMAGIC()` since their implementation handles ‘get’ magic.

Finding Magic

```
MAGIC* mg_find(SV*, int type); /* Finds the magic pointer of that type */
```

This routine returns a pointer to the `MAGIC` structure stored in the SV. If the SV does not have that magical feature, `NULL` is returned. Also, if the SV is not of type `SVt_PVMG`, Perl may core dump.

```
int mg_copy(SV* sv, SV* nsv, const char* key, STRLEN klen);
```

This routine checks to see what types of magic `sv` has. If the `mg_type` field is an uppercase letter, then the `mg_obj` is copied to `nsv`, but the `mg_type` field is changed to be the lowercase letter.

Understanding the Magic of Tied Hashes and Arrays

Tied hashes and arrays are magical beasts of the `PERL_MAGIC_tied` magic type.

WARNING: As of the 5.004 release, proper usage of the array and hash access functions requires understanding a few caveats. Some of these caveats are actually considered bugs in the API, to be fixed in later releases, and are bracketed with `[MAYCHANGE]` below. If you find yourself actually applying such information in this section, be aware that the behavior may change in the future, umm, without warning.

The `perl tie` function associates a variable with an object that implements the various `GET`, `SET`, etc methods. To perform the equivalent of the `perl tie` function from an XSUB, you must mimic this behaviour. The code below carries out the necessary steps – firstly it creates a new hash, and then creates a second hash which it blesses into the class which will implement the tie methods. Lastly it ties the two hashes together, and returns a reference to the new tied hash. Note that the code below does **NOT** call the `TIEHASH` method in the `MyTie` class – see [Calling Perl Routines from within C Programs](#) for details on how to do this.

```
SV*
mytie()
PREINIT:
    HV *hash;
    HV *stash;
    SV *tie;
CODE:
    hash = newHV();
    tie = newRV_noinc((SV*)newHV());
    stash = gv_stashpv("MyTie", TRUE);
    sv_bless(tie, stash);
    hv_magic(hash, (GV*)tie, PERL_MAGIC_tied);
    RETVAL = newRV_noinc(hash);
OUTPUT:
    RETVAL
```

The `av_store` function, when given a tied array argument, merely copies the magic of the array onto the value to be “stored”, using `mg_copy`. It may also return `NULL`, indicating that the value did not actually need to be stored in the array. `[MAYCHANGE]` After a call to `av_store` on a tied array, the caller will usually need to call `mg_set(val)` to actually invoke the perl level “STORE” method on the `TIEARRAY` object. If `av_store` did return `NULL`, a call to `SvREFCNT_dec(val)` will also be usually necessary to avoid a memory leak. `[MAYCHANGE]`

The previous paragraph is applicable verbatim to tied hash access using the `hv_store` and `hv_store_ent` functions as well.

`av_fetch` and the corresponding hash functions `hv_fetch` and `hv_fetch_ent` actually return an undefined mortal value whose magic has been initialized using `mg_copy`. Note the value so returned does not need to be deallocated, as it is already mortal. [MAYCHANGE] But you will need to call `mg_get()` on the returned value in order to actually invoke the perl level "FETCH" method on the underlying TIE object. Similarly, you may also call `mg_set()` on the return value after possibly assigning a suitable value to it using `sv_setsv`, which will invoke the "STORE" method on the TIE object. [/MAYCHANGE]

[MAYCHANGE] In other words, the array or hash fetch/store functions don't really fetch and store actual values in the case of tied arrays and hashes. They merely call `mg_copy` to attach magic to the values that were meant to be "stored" or "fetched". Later calls to `mg_get` and `mg_set` actually do the job of invoking the TIE methods on the underlying objects. Thus the magic mechanism currently implements a kind of lazy access to arrays and hashes.

Currently (as of perl version 5.004), use of the hash and array access functions requires the user to be aware of whether they are operating on "normal" hashes and arrays, or on their tied variants. The API may be changed to provide more transparent access to both tied and normal data types in future versions. [/MAYCHANGE]

You would do well to understand that the TIEARRAY and TIEHASH interfaces are mere sugar to invoke some perl method calls while using the uniform hash and array syntax. The use of this sugar imposes some overhead (typically about two to four extra opcodes per FETCH/STORE operation, in addition to the creation of all the mortal variables required to invoke the methods). This overhead will be comparatively small if the TIE methods are themselves substantial, but if they are only a few statements long, the overhead will not be insignificant.

Localizing changes

Perl has a very handy construction

```
{
    local $var = 2;
    ...
}
```

This construction is *approximately* equivalent to

```
{
    my $oldvar = $var;
    $var = 2;
    ...
    $var = $oldvar;
}
```

The biggest difference is that the first construction would reinstate the initial value of `$var`, irrespective of how control exits the block: `goto`, `return`, `die/eval`, etc. It is a little bit more efficient as well.

There is a way to achieve a similar task from C via Perl API: create a *pseudo-block*, and arrange for some changes to be automatically undone at the end of it, either explicit, or via a non-local exit (via `die()`). A *block*-like construct is created by a pair of ENTER/LEAVE macros (see [Returning a Scalar in perlcall](#)). Such a construct may be created specially for some important localized task, or an existing one (like boundaries of enclosing Perl subroutine/block, or an existing pair for freeing TMPs) may be used. (In the second case the overhead of additional localization must be almost negligible.) Note that any XSUB is automatically enclosed in an ENTER/LEAVE pair.

Inside such a *pseudo-block* the following service is available:

SAVEINT(int i)

SAVEIV(IV i)

SAVEI32(I32 i)

SAVELONG(long i)

These macros arrange things to restore the value of integer variable *i* at the end of enclosing *pseudo-block*.

SAVESPTR(s)

SAVEPPTR(p)

These macros arrange things to restore the value of pointers *s* and *p*. *s* must be a pointer of a type which survives conversion to *SV** and back, *p* should be able to survive conversion to *char** and back.

SAVEFREESV(SV *sv)

The refcount of *sv* would be decremented at the end of *pseudo-block*. This is similar to *sv_2mortal* in that it is also a mechanism for doing a delayed *SvREFCNT_dec*. However, while *sv_2mortal* extends the lifetime of *sv* until the beginning of the next statement, *SAVEFREESV* extends it until the end of the enclosing scope. These lifetimes can be wildly different.

Also compare *SAVEMORTALIZESV*.

SAVEMORTALIZESV(SV *sv)

Just like *SAVEFREESV*, but mortalizes *sv* at the end of the current scope instead of decrementing its reference count. This usually has the effect of keeping *sv* alive until the statement that called the currently live scope has finished executing.

SAVEFREEOP(OP *op)

The *OP ** is *op_free()* ed at the end of *pseudo-block*.

SAVEFREEPV(p)

The chunk of memory which is pointed to by *p* is *Safefree()* ed at the end of *pseudo-block*.

SAVECLEARSV(SV *sv)

Clears a slot in the current scratchpad which corresponds to *sv* at the end of *pseudo-block*.

SAVEDELETE(HV *hv, char *key, I32 length)

The key *key* of *hv* is deleted at the end of *pseudo-block*. The string pointed to by *key* is *Safefree()* ed. If one has a *key* in short-lived storage, the corresponding string may be reallocated like this:

```
SAVEDELETE(PL_defstash, savepv(tmpbuf), strlen(tmpbuf));
```

SAVEDESTRUCTOR(DESTRUCTORFUNC_NOCONTEXT_t f, void *p)

At the end of *pseudo-block* the function *f* is called with the only argument *p*.

SAVEDESTRUCTOR_X(DESTRUCTORFUNC_t f, void *p)

At the end of *pseudo-block* the function *f* is called with the implicit context argument (if any), and *p*.

SAVESTACK_POS()

The current offset on the Perl internal stack (cf. *SP*) is restored at the end of *pseudo-block*.

The following API list contains functions, thus one needs to provide pointers to the modifiable data explicitly (either C pointers, or Perlish *GV **s). Where the above macros take *int*, a similar function takes *int **.

SV* save_scalar(GV *gv)

Equivalent to Perl code *local \$gv*.

AV* `save_ary(GV *gv)`

HV* `save_hash(GV *gv)`

Similar to `save_scalar`, but localize `@gv` and `%gv`.

`void save_item(SV *item)`

Duplicates the current value of SV, on the exit from the current *ENTER/LEAVE pseudo-block* will restore the value of SV using the stored value.

`void save_list(SV **sarg, I32 maxsarg)`

A variant of `save_item` which takes multiple arguments via an array `sarg` of SV* of length `maxsarg`.

SV* `save_svref(SV **sptr)`

Similar to `save_scalar`, but will reinstate an SV *.

`void save_aptr(AV **aptr)`

`void save_hptr(HV **hptr)`

Similar to `save_svref`, but localize AV * and HV *.

The `Alias` module implements localization of the basic types within the *caller's scope*. People who are interested in how to localize things in the containing scope should take a look there too.

Subroutines

XSUBs and the Argument Stack

The XSUB mechanism is a simple way for Perl programs to access C subroutines. An XSUB routine will have a stack that contains the arguments from the Perl program, and a way to map from the Perl data structures to a C equivalent.

The stack arguments are accessible through the `ST(n)` macro, which returns the *n*'th stack argument. Argument 0 is the first argument passed in the Perl subroutine call. These arguments are SV*, and can be used anywhere an SV* is used.

Most of the time, output from the C routine can be handled through use of the `RETVAl` and `OUTPUT` directives. However, there are some cases where the argument stack is not already long enough to handle all the return values. An example is the `POSIX tzname()` call, which takes no arguments, but returns two, the local time zone's standard and summer time abbreviations.

To handle this situation, the `PPCODE` directive is used and the stack is extended using the macro:

```
EXTEND(SP, num);
```

where `SP` is the macro that represents the local copy of the stack pointer, and `num` is the number of elements the stack should be extended by.

Now that there is room on the stack, values can be pushed on it using `PUSHs` macro. The values pushed will often need to be "mortal" (See [/Reference Counts and Mortality](#)).

```
PUSHs(sv_2mortal(newSViv(an_integer)))
PUSHs(sv_2mortal(newSVpv("Some String", 0)))
PUSHs(sv_2mortal(newSVnv(3.141592)))
```

And now the Perl program calling `tzname`, the two values will be assigned as in:

```
($standard_abbrev, $summer_abbrev) = POSIX::tzname;
```

An alternate (and possibly simpler) method to pushing values on the stack is to use the macro:

```
XPUSHs(SV*)
```

This macro automatically adjust the stack for you, if needed. Thus, you do not need to call `EXTEND` to extend the stack.

Despite their suggestions in earlier versions of this document the macros `PUSHi`, `PUSHn` and `PUSHp` are *not* suited to XSUBs which return multiple results, see */Putting a C value on Perl stack*.

For more information, consult [perlxs](#) and [perlxsut](#).

Calling Perl Routines from within C Programs

There are four routines that can be used to call a Perl subroutine from within a C program. These four are:

```
I32  call_sv(SV*, I32);
I32  call_pv(const char*, I32);
I32  call_method(const char*, I32);
I32  call_argv(const char*, I32, register char**);
```

The routine most often used is `call_sv`. The `SV*` argument contains either the name of the Perl subroutine to be called, or a reference to the subroutine. The second argument consists of flags that control the context in which the subroutine is called, whether or not the subroutine is being passed arguments, how errors should be trapped, and how to treat return values.

All four routines return the number of arguments that the subroutine returned on the Perl stack.

These routines used to be called `perl_call_sv`, etc., before Perl v5.6.0, but those names are now deprecated; macros of the same name are provided for compatibility.

When using any of these routines (except `call_argv`), the programmer must manipulate the Perl stack. These include the following macros and functions:

```
dSP
SP
PUSHMARK()
PUTBACK
SPAGAIN
ENTER
SAVETMPS
FREETMPS
LEAVE
XPUSH*()
POP*()
```

For a detailed description of calling conventions from C to Perl, consult [perlcall](#).

Memory Allocation

All memory meant to be used with the Perl API functions should be manipulated using the macros described in this section. The macros provide the necessary transparency between differences in the actual malloc implementation that is used within perl.

It is suggested that you enable the version of malloc that is distributed with Perl. It keeps pools of various sizes of unallocated memory in order to satisfy allocation requests more quickly. However, on some platforms, it may cause spurious malloc or free errors.

```
New(x, pointer, number, type);
Newc(x, pointer, number, type, cast);
Newz(x, pointer, number, type);
```

These three macros are used to initially allocate memory.

The first argument `x` was a "magic cookie" that was used to keep track of who called the macro, to help when debugging memory problems. However, the current code makes no use of this feature (most Perl developers now use run-time memory checkers), so this argument can be any number.

The second argument `pointer` should be the name of a variable that will point to the newly allocated memory.

The third and fourth arguments `number` and `type` specify how many of the specified type of data structure should be allocated. The argument `type` is passed to `sizeof`. The final argument to `Newc`, `cast`, should be used if the `pointer` argument is different from the `type` argument.

Unlike the `New` and `Newc` macros, the `Newz` macro calls `memzero` to zero out all the newly allocated memory.

```
Renew(pointer, number, type);
Renewc(pointer, number, type, cast);
Safefree(pointer)
```

These three macros are used to change a memory buffer size or to free a piece of memory no longer needed. The arguments to `Renew` and `Renewc` match those of `New` and `Newc` with the exception of not needing the "magic cookie" argument.

```
Move(source, dest, number, type);
Copy(source, dest, number, type);
Zero(dest, number, type);
```

These three macros are used to move, copy, or zero out previously allocated memory. The `source` and `dest` arguments point to the source and destination starting points. Perl will move, copy, or zero out `number` instances of the size of the `type` data structure (using the `sizeof` function).

PerlIO

The most recent development releases of Perl has been experimenting with removing Perl's dependency on the "normal" standard I/O suite and allowing other stdio implementations to be used. This involves creating a new abstraction layer that then calls whichever implementation of stdio Perl was compiled with. All XSUBs should now use the functions in the PerlIO abstraction layer and not make any assumptions about what kind of stdio is being used.

For a complete description of the PerlIO abstraction, consult [perlapi0](#).

Putting a C value on Perl stack

A lot of opcodes (this is an elementary operation in the internal perl stack machine) put an SV* on the stack. However, as an optimization the corresponding SV is (usually) not recreated each time. The opcodes reuse specially assigned SVs (*targets*) which are (as a corollary) not constantly freed/created.

Each of the targets is created only once (but see [Scratchpads and recursion](#) below), and when an opcode needs to put an integer, a double, or a string on stack, it just sets the corresponding parts of its *target* and puts the *target* on stack.

The macro to put this target on stack is `PUSHTARG`, and it is directly used in some opcodes, as well as indirectly in zillions of others, which use it via `(X) PUSH [pni]`.

Because the target is reused, you must be careful when pushing multiple values on the stack. The following code will not do what you think:

```
XPUSHi(10);
XPUSHi(20);
```

This translates as "set TARG to 10, push a pointer to TARG onto the stack; set TARG to 20, push a pointer to TARG onto the stack". At the end of the operation, the stack does not contain the values 10 and 20, but actually contains two pointers to TARG, which we have set to 20. If you need to push multiple different values, use `XPUSHs`, which bypasses TARG.

On a related note, if you do use `(X) PUSH [pni]`, then you're going to need a `dTARG` in your variable declarations so that the `*PUSH*` macros can make use of the local variable TARG.

Scratchpads

The question remains on when the SVs which are *targets* for opcodes are created. The answer is that they are created when the current unit — a subroutine or a file (for opcodes for statements outside of subroutines) —

is compiled. During this time a special anonymous Perl array is created, which is called a scratchpad for the current unit.

A scratchpad keeps SVs which are lexicals for the current unit and are targets for opcodes. One can deduce that an SV lives on a scratchpad by looking on its flags: lexicals have `SVs_PADMY` set, and *targets* have `SVs_PADTMP` set.

The correspondence between OPs and *targets* is not 1-to-1. Different OPs in the compile tree of the unit can use the same target, if this would not conflict with the expected life of the temporary.

Scratchpads and recursion

In fact it is not 100% true that a compiled unit contains a pointer to the scratchpad AV. In fact it contains a pointer to an AV of (initially) one element, and this element is the scratchpad AV. Why do we need an extra level of indirection?

The answer is **recursion**, and maybe **threads**. Both these can create several execution pointers going into the same subroutine. For the subroutine-child not write over the temporaries for the subroutine-parent (lifespan of which covers the call to the child), the parent and the child should have different scratchpads. (*And* the lexicals should be separate anyway!)

So each subroutine is born with an array of scratchpads (of length 1). On each entry to the subroutine it is checked that the current depth of the recursion is not more than the length of this array, and if it is, new scratchpad is created and pushed into the array.

The *targets* on this scratchpad are undefs, but they are already marked with correct flags.

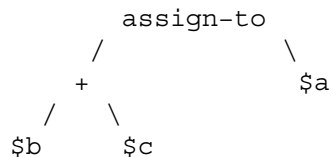
Compiled code

Code tree

Here we describe the internal form your code is converted to by Perl. Start with a simple example:

```
$a = $b + $c;
```

This is converted to a tree similar to this one:



(but slightly more complicated). This tree reflects the way Perl parsed your code, but has nothing to do with the execution order. There is an additional "thread" going through the nodes of the tree which shows the order of execution of the nodes. In our simplified example above it looks like:

```
$b ----> $c ----> + ----> $a ----> assign-to
```

But with the actual compile tree for `$a = $b + $c` it is different: some nodes *optimized away*. As a corollary, though the actual tree contains more nodes than our simplified example, the execution order is the same as in our example.

Examining the tree

If you have your perl compiled for debugging (usually done with `-D optimize=-g` on Configure command line), you may examine the compiled tree by specifying `-Dx` on the Perl command line. The output takes several lines per node, and for `$b+$c` it looks like this:

```

5          TYPE = add   ==> 6
          TARG = 1
          FLAGS = (SCALAR, KIDS)
          {
              TYPE = null ==> (4)
              (was rv2sv)

```

```

        FLAGS = (SCALAR,KIDS)
        {
3           TYPE = gvsv ==> 4
            FLAGS = (SCALAR)
            GV = main::b
        }
    }
    {
        TYPE = null ==> (5)
        (was rv2sv)
        FLAGS = (SCALAR,KIDS)
        {
4           TYPE = gvsv ==> 5
            FLAGS = (SCALAR)
            GV = main::c
        }
    }
}

```

This tree has 5 nodes (one per TYPE specifier), only 3 of them are not optimized away (one per number in the left column). The immediate children of the given node correspond to `{ }` pairs on the same level of indentation, thus this listing corresponds to the tree:

```

      add
     /  \
  null  null
   |    |
  gvsv  gvsv

```

The execution order is indicated by `==>` marks, thus it is 3 4 5 6 (node 6 is not included into above listing), i.e., `gvsv gvsv add whatever`.

Each of these nodes represents an op, a fundamental operation inside the Perl core. The code which implements each operation can be found in the *pp*.c* files; the function which implements the op with type `gvsv` is `pp_gvsv`, and so on. As the tree above shows, different ops have different numbers of children: `add` is a binary operator, as one would expect, and so has two children. To accommodate the various different numbers of children, there are various types of op data structure, and they link together in different ways.

The simplest type of op structure is `OP`: this has no children. Unary operators, `UNOPs`, have one child, and this is pointed to by the `op_first` field. Binary operators (`BINOPs`) have not only an `op_first` field but also an `op_last` field. The most complex type of op is a `LISTOP`, which has any number of children. In this case, the first child is pointed to by `op_first` and the last child by `op_last`. The children in between can be found by iteratively following the `op_sibling` pointer from the first child to the last.

There are also two other op types: a `PMOP` holds a regular expression, and has no children, and a `LOOP` may or may not have children. If the `op_children` field is non-zero, it behaves like a `LISTOP`. To complicate matters, if a `UNOP` is actually a null op after optimization (see [/Compile pass 2: context propagation](#)) it will still have children in accordance with its former type.

Compile pass 1: check routines

The tree is created by the compiler while *yacc* code feeds it the constructions it recognizes. Since *yacc* works bottom-up, so does the first pass of perl compilation.

What makes this pass interesting for perl developers is that some optimization may be performed on this pass. This is optimization by so-called "check routines". The correspondence between node names and corresponding check routines is described in *opcode.pl* (do not forget to run `make regen_headers` if you modify this file).

A check routine is called when the node is fully constructed except for the execution-order thread. Since at this time there are no back-links to the currently constructed node, one can do most any operation to the top-level node, including freeing it and/or creating new nodes above/below it.

The check routine returns the node which should be inserted into the tree (if the top-level node was not modified, check routine returns its argument).

By convention, check routines have names `ck_*`. They are usually called from `new*OP` subroutines (or `convert`) (which in turn are called from *perly.y*).

Compile pass 1a: constant folding

Immediately after the check routine is called the returned node is checked for being compile-time executable. If it is (the value is judged to be constant) it is immediately executed, and a *constant* node with the "return value" of the corresponding subtree is substituted instead. The subtree is deleted.

If constant folding was not performed, the execution-order thread is created.

Compile pass 2: context propagation

When a context for a part of compile tree is known, it is propagated down through the tree. At this time the context can have 5 values (instead of 2 for runtime context): void, boolean, scalar, list, and lvalue. In contrast with the pass 1 this pass is processed from top to bottom: a node's context determines the context for its children.

Additional context-dependent optimizations are performed at this time. Since at this moment the compile tree contains back-references (via "thread" pointers), nodes cannot be `free()`d now. To allow optimized-away nodes at this stage, such nodes are `null()`ified instead of `free()`ing (i.e. their type is changed to `OP_NULL`).

Compile pass 3: peephole optimization

After the compile tree for a subroutine (or for an `eval` or a file) is created, an additional pass over the code is performed. This pass is neither top-down or bottom-up, but in the execution order (with additional complications for conditionals). These optimizations are done in the subroutine `peep()`. Optimizations performed at this stage are subject to the same restrictions as in the pass 2.

Pluggable runops

The compile tree is executed in a runops function. There are two runops functions in *run.c*. `Perl_runops_debug` is used with `DEBUGGING` and `Perl_runops_standard` is used otherwise. For fine control over the execution of the compile tree it is possible to provide your own runops function.

It's probably best to copy one of the existing runops functions and change it to suit your needs. Then, in the `BOOT` section of your XS file, add the line:

```
PL_runops = my_runops;
```

This function should be as efficient as possible to keep your programs running as fast as possible.

Examining internal data structures with the `dump` functions

To aid debugging, the source file *dump.c* contains a number of functions which produce formatted output of internal data structures.

The most commonly used of these functions is `Perl_sv_dump`; it's used for dumping SVs, AVs, HVs, and CVs. The `Devel::Peek` module calls `sv_dump` to produce debugging output from Perl-space, so users of that module should already be familiar with its format.

`Perl_op_dump` can be used to dump an OP structure or any of its derivatives, and produces output similar to `perl -Dx`; in fact, `Perl_dump_eval` will dump the main root of the code being evaluated, exactly like `-Dx`.

Other useful functions are `Perl_dump_sub`, which turns a GV into an op tree, `Perl_dump_packsubs` which calls `Perl_dump_sub` on all the subroutines in a package like so: (Thankfully, these are all xsubs, so there is no op tree)

```
(gdb) print Perl_dump_packsubs(PL_defstash)
SUB attributes::bootstrap = (xsub 0x811fedc 0)
SUB UNIVERSAL::can = (xsub 0x811f50c 0)
SUB UNIVERSAL::isa = (xsub 0x811f304 0)
SUB UNIVERSAL::VERSION = (xsub 0x811f7ac 0)
SUB DynaLoader::boot_DynaLoader = (xsub 0x805b188 0)
```

and `Perl_dump_all`, which dumps all the subroutines in the stash and the op tree of the main root.

How multiple interpreters and concurrency are supported

Background and PERL_IMPLICIT_CONTEXT

The Perl interpreter can be regarded as a closed box: it has an API for feeding it code or otherwise making it do things, but it also has functions for its own use. This smells a lot like an object, and there are ways for you to build Perl so that you can have multiple interpreters, with one interpreter represented either as a C structure, or inside a thread-specific structure. These structures contain all the context, the state of that interpreter.

Two macros control the major Perl build flavors: `MULTIPLICITY` and `USE_5005THREADS`. The `MULTIPLICITY` build has a C structure that packages all the interpreter state, and there is a similar thread-specific data structure under `USE_5005THREADS`. In both cases, `PERL_IMPLICIT_CONTEXT` is also normally defined, and enables the support for passing in a "hidden" first argument that represents all three data structures.

All this obviously requires a way for the Perl internal functions to be either subroutines taking some kind of structure as the first argument, or subroutines taking nothing as the first argument. To enable these two very different ways of building the interpreter, the Perl source (as it does in so many other situations) makes heavy use of macros and subroutine naming conventions.

First problem: deciding which functions will be public API functions and which will be private. All functions whose names begin `S_` are private (think "S" for "secret" or "static"). All other functions begin with `"Perl_"`, but just because a function begins with `"Perl_"` does not mean it is part of the API. (See [/Internal Functions](#).) The easiest way to be **sure** a function is part of the API is to find its entry in [perlapi](#). If it exists in [perlapi](#), it's part of the API. If it doesn't, and you think it should be (i.e., you need it for your extension), send mail via [perlbug](#) explaining why you think it should be.

Second problem: there must be a syntax so that the same subroutine declarations and calls can pass a structure as their first argument, or pass nothing. To solve this, the subroutines are named and declared in a particular way. Here's a typical start of a static function used within the Perl guts:

```
STATIC void
S_incline(pTHX_ char *s)
```

`STATIC` becomes "static" in C, and may be `#define'd` to nothing in some configurations in future.

A public function (i.e. part of the internal API, but not necessarily sanctioned for use in extensions) begins like this:

```
void
Perl_sv_setsv(pTHX_ SV* dsv, SV* ssv)
```

`pTHX_` is one of a number of macros (in `perl.h`) that hide the details of the interpreter's context. `THX` stands for "thread", "this", or "thingy", as the case may be. (And no, George Lucas is not involved. :-) The first character could be 'p' for a **p**rototype, 'a' for **a**rgument, or 'd' for **d**eclaration, so we have `pTHX`, `aTHX` and `dTHX`, and their variants.

When Perl is built without options that set `PERL_IMPLICIT_CONTEXT`, there is no first argument containing the interpreter's context. The trailing underscore in the `pTHX_` macro indicates that the macro

expansion needs a comma after the context argument because other arguments follow it. If `PERL_IMPLICIT_CONTEXT` is not defined, `pTHX_` will be ignored, and the subroutine is not prototyped to take the extra argument. The form of the macro without the trailing underscore is used when there are no additional explicit arguments.

When a core function calls another, it must pass the context. This is normally hidden via macros. Consider `sv_setsv`. It expands into something like this:

```

#ifdef PERL_IMPLICIT_CONTEXT
    define sv_setsv(a,b)          Perl_sv_setsv(aTHX_ a, b)
    /* can't do this for vararg functions, see below */
else
    define sv_setsv              Perl_sv_setsv
#endif

```

This works well, and means that XS authors can gleefully write:

```
sv_setsv(foo, bar);
```

and still have it work under all the modes Perl could have been compiled with.

This doesn't work so cleanly for varargs functions, though, as macros imply that the number of arguments is known in advance. Instead we either need to spell them out fully, passing `aTHX_` as the first argument (the Perl core tends to do this with functions like `Perl_warner`), or use a context-free version.

The context-free version of `Perl_warner` is called `Perl_warner_nocontext`, and does not take the extra argument. Instead it does `dTHX`; to get the context from thread-local storage. We `#define warner Perl_warner_nocontext` so that extensions get source compatibility at the expense of performance. (Passing an arg is cheaper than grabbing it from thread-local storage.)

You can ignore `[pad]THXx` when browsing the Perl headers/sources. Those are strictly for use within the core. Extensions and embedders need only be aware of `[pad]THX`.

So what happened to dTHR?

`dTHR` was introduced in perl 5.005 to support the older thread model. The older thread model now uses the `THX` mechanism to pass context pointers around, so `dTHR` is not useful any more. Perl 5.6.0 and later still have it for backward source compatibility, but it is defined to be a no-op.

How do I use all this in extensions?

When Perl is built with `PERL_IMPLICIT_CONTEXT`, extensions that call any functions in the Perl API will need to pass the initial context argument somehow. The kicker is that you will need to write it in such a way that the extension still compiles when Perl hasn't been built with `PERL_IMPLICIT_CONTEXT` enabled.

There are three ways to do this. First, the easy but inefficient way, which is also the default, in order to maintain source compatibility with extensions: whenever `XSUB.h` is `#included`, it redefines the `aTHX` and `aTHX_` macros to call a function that will return the context. Thus, something like:

```
sv_setsv(asv, bsv);
```

in your extension will translate to this when `PERL_IMPLICIT_CONTEXT` is in effect:

```
Perl_sv_setsv(Perl_get_context(), asv, bsv);
```

or to this otherwise:

```
Perl_sv_setsv(asv, bsv);
```

You have to do nothing new in your extension to get this; since the Perl library provides `Perl_get_context()`, it will all just work.

The second, more efficient way is to use the following template for your `Foo.xs`:

```

#define PERL_NO_GET_CONTEXT      /* we want efficiency */
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

static my_private_function(int arg1, int arg2);

static SV *
my_private_function(int arg1, int arg2)
{
    dTHX;          /* fetch context */
    ... call many Perl API functions ...
}

[... etc ...]

MODULE = Foo          PACKAGE = Foo

/* typical XSUB */

void
my_xsub(arg)
    int arg
    CODE:
        my_private_function(arg, 10);

```

Note that the only two changes from the normal way of writing an extension is the addition of a `#define PERL_NO_GET_CONTEXT` before including the Perl headers, followed by a `dTHX;` declaration at the start of every function that will call the Perl API. (You'll know which functions need this, because the C compiler will complain that there's an undeclared identifier in those functions.) No changes are needed for the XSUBs themselves, because the `XS()` macro is correctly defined to pass in the implicit context if needed.

The third, even more efficient way is to ape how it is done within the Perl guts:

```

#define PERL_NO_GET_CONTEXT      /* we want efficiency */
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

/* pTHX_ only needed for functions that call Perl API */
static my_private_function(pTHX_ int arg1, int arg2);

static SV *
my_private_function(pTHX_ int arg1, int arg2)
{
    /* dTHX; not needed here, because THX is an argument */
    ... call Perl API functions ...
}

[... etc ...]

MODULE = Foo          PACKAGE = Foo

/* typical XSUB */

void
my_xsub(arg)
    int arg
    CODE:
        my_private_function(aTHX_ arg, 10);

```

This implementation never has to fetch the context using a function call, since it is always passed as an extra argument. Depending on your needs for simplicity or efficiency, you may mix the previous two approaches freely.

Never add a comma after `pTHX` yourself—always use the form of the macro with the underscore for functions that take explicit arguments, or the form without the argument for functions with no explicit arguments.

Should I do anything special if I call perl from multiple threads?

If you create interpreters in one thread and then proceed to call them in another, you need to make sure perl's own Thread Local Storage (TLS) slot is initialized correctly in each of those threads.

The `perl_alloc` and `perl_clone` API functions will automatically set the TLS slot to the interpreter they created, so that there is no need to do anything special if the interpreter is always accessed in the same thread that created it, and that thread did not create or call any other interpreters afterwards. If that is not the case, you have to set the TLS slot of the thread before calling any functions in the Perl API on that particular interpreter. This is done by calling the `PERL_SET_CONTEXT` macro in that thread as the first thing you do:

```
/* do this before doing anything else with some_perl */
PERL_SET_CONTEXT(some_perl);

... other Perl API calls on some_perl go here ...
```

Future Plans and `PERL_IMPLICIT_SYS`

Just as `PERL_IMPLICIT_CONTEXT` provides a way to bundle up everything that the interpreter knows about itself and pass it around, so too are there plans to allow the interpreter to bundle up everything it knows about the environment it's running on. This is enabled with the `PERL_IMPLICIT_SYS` macro. Currently it only works with `USE_ITHREADS` and `USE_5005THREADS` on Windows (see inside `iperlsys.h`).

This allows the ability to provide an extra pointer (called the "host" environment) for all the system calls. This makes it possible for all the system stuff to maintain their own state, broken down into seven C structures. These are thin wrappers around the usual system calls (see `win32/perl-lib.c`) for the default perl executable, but for a more ambitious host (like the one that would do `fork()` emulation) all the extra work needed to pretend that different interpreters are actually different "processes", would be done here.

The Perl engine/interpreter and the host are orthogonal entities. There could be one or more interpreters in a process, and one or more "hosts", with free association between them.

Internal Functions

All of Perl's internal functions which will be exposed to the outside world are prefixed by `Perl_` so that they will not conflict with XS functions or functions used in a program in which Perl is embedded. Similarly, all global variables begin with `PL_`. (By convention, static functions start with `S_`)

Inside the Perl core, you can get at the functions either with or without the `Perl_` prefix, thanks to a bunch of defines that live in ***embed.h***. This header file is generated automatically from ***embed.pl***. ***embed.pl*** also creates the prototyping header files for the internal functions, generates the documentation and a lot of other bits and pieces. It's important that when you add a new function to the core or change an existing one, you change the data in the table at the end of ***embed.pl*** as well. Here's a sample entry from that table:

```
Apd |SV**    |av_fetch    |AV* ar|I32 key|I32 lval
```

The second column is the return type, the third column the name. Columns after that are the arguments. The first column is a set of flags:

A This function is a part of the public API.

p This function has a `Perl_` prefix; ie, it is defined as `Perl_av_fetch`

d This function has documentation using the apidoc feature which we'll look at in a second.

Other available flags are:

s This is a static function and is defined as `S_whatever`, and usually called within the sources as `whatever(...)`.

n This does not use `aTHX_` and `pTHX` to pass interpreter context. (See [Background and PERL_IMPLICIT_CONTEXT](#).)

r This function never returns; `croak`, `exit` and friends.

f This function takes a variable number of arguments, `printf` style. The argument list should end with `...`, like this:

```
Afprd    |void    |croak                |const char* pat|...
```

M This function is part of the experimental development API, and may change or disappear without notice.

o This function should not have a compatibility macro to define, say, `Perl_parse` to `parse`. It must be called as `Perl_parse`.

j This function is not a member of `CPerlObj`. If you don't know what this means, don't use it.

x This function isn't exported out of the Perl core.

If you edit *embed.pl*, you will need to run `make regen_headers` to force a rebuild of *embed.h* and other auto-generated files.

Formatted Printing of IVs, UVs, and NVs

If you are printing IVs, UVs, or NVs instead of the `stdio(3)` style formatting codes like `%d`, `%ld`, `%f`, you should use the following macros for portability

<code>IVdf</code>	IV in decimal
<code>UVuf</code>	UV in decimal
<code>UVof</code>	UV in octal
<code>UVxf</code>	UV in hexadecimal
<code>NVef</code>	NV %e-like
<code>NVff</code>	NV %f-like
<code>NVgf</code>	NV %g-like

These will take care of 64-bit integers and long doubles. For example:

```
printf("IV is %"IVdf"\n", iv);
```

The `IVdf` will expand to whatever is the correct format for the IVs.

If you are printing addresses of pointers, use `UVxf` combined with `PTR2UV()`, do not use `%lx` or `%p`.

Pointer-To-Integer and Integer-To-Pointer

Because pointer size does not necessarily equal integer size, use the follow macros to do it right.

```
PTR2UV(pointer)
PTR2IV(pointer)
PTR2NV(pointer)
INT2PTR(pointertype, integer)
```

For example:

```
IV iv = ...;
SV *sv = INT2PTR(SV*, iv);
```

and

```
AV *av = ...;
UV uv = PTR2UV(av);
```

Source Documentation

There's an effort going on to document the internal functions and automatically produce reference manuals from them – *perlapi* is one such manual which details all the functions which are available to XS writers. *perlintern* is the autogenerated manual for the functions which are not part of the API and are supposedly for internal use only.

Source documentation is created by putting POD comments into the C source, like this:

```
/*
=for apidoc sv_setiv

Copies an integer into the given SV. Does not handle 'set' magic. See
C<sv_setiv_mg>.

=cut
*/
```

Please try and supply some documentation if you add functions to the Perl core.

Unicode Support

Perl 5.6.0 introduced Unicode support. It's important for porters and XS writers to understand this support and make sure that the code they write does not corrupt Unicode data.

What is Unicode, anyway?

In the olden, less enlightened times, we all used to use ASCII. Most of us did, anyway. The big problem with ASCII is that it's American. Well, no, that's not actually the problem; the problem is that it's not particularly useful for people who don't use the Roman alphabet. What used to happen was that particular languages would stick their own alphabet in the upper range of the sequence, between 128 and 255. Of course, we then ended up with plenty of variants that weren't quite ASCII, and the whole point of it being a standard was lost.

Worse still, if you've got a language like Chinese or Japanese that has hundreds or thousands of characters, then you really can't fit them into a mere 256, so they had to forget about ASCII altogether, and build their own systems using pairs of numbers to refer to one character.

To fix this, some people formed Unicode, Inc. and produced a new character set containing all the characters you can possibly think of and more. There are several ways of representing these characters, and the one Perl uses is called UTF8. UTF8 uses a variable number of bytes to represent a character, instead of just one. You can learn more about Unicode at <http://www.unicode.org/>

How can I recognise a UTF8 string?

You can't. This is because UTF8 data is stored in bytes just like non-UTF8 data. The Unicode character 200, (0xC8 for you hex types) capital E with a grave accent, is represented by the two bytes `v196.172`. Unfortunately, the non-Unicode string `chr(196).chr(172)` has that byte sequence as well. So you can't tell just by looking – this is what makes Unicode input an interesting problem.

The API function `is_utf8_string` can help; it'll tell you if a string contains only valid UTF8 characters. However, it can't do the work for you. On a character-by-character basis, `is_utf8_char` will tell you whether the current character in a string is valid UTF8.

How does UTF8 represent Unicode characters?

As mentioned above, UTF8 uses a variable number of bytes to store a character. Characters with values 1...128 are stored in one byte, just like good ol' ASCII. Character 129 is stored as `v194.129`; this continues up to character 191, which is `v194.191`. Now we've run out of bits (191 is binary 10111111) so we move on; 192 is `v195.128`. And so it goes on, moving to three bytes at character 2048.

Assuming you know you're dealing with a UTF8 string, you can find out how long the first character in it is with the `UTF8SKIP` macro:

```
char *utf = "\305\233\340\240\201";
I32 len;

len = UTF8SKIP(utf); /* len is 2 here */
utf += len;
len = UTF8SKIP(utf); /* len is 3 here */
```

Another way to skip over characters in a UTF8 string is to use `utf8_hop`, which takes a string and a number of characters to skip over. You're on your own about bounds checking, though, so don't use it lightly.

All bytes in a multi-byte UTF8 character will have the high bit set, so you can test if you need to do something special with this character like this:

```
UV uv;

if (utf & 0x80)
    /* Must treat this as UTF8 */
    uv = utf8_to_uv(utf);
else
    /* OK to treat this character as a byte */
    uv = *utf;
```

You can also see in that example that we use `utf8_to_uv` to get the value of the character; the inverse function `uv_to_utf8` is available for putting a UV into UTF8:

```
if (uv > 0x80)
    /* Must treat this as UTF8 */
    utf8 = uv_to_utf8(utf8, uv);
else
    /* OK to treat this character as a byte */
    *utf8++ = uv;
```

You **must** convert characters to UVs using the above functions if you're ever in a situation where you have to match UTF8 and non-UTF8 characters. You may not skip over UTF8 characters in this case. If you do this, you'll lose the ability to match hi-bit non-UTF8 characters; for instance, if your UTF8 string contains `v196.172`, and you skip that character, you can never match a `chr(200)` in a non-UTF8 string. So don't do that!

How does Perl store UTF8 strings?

Currently, Perl deals with Unicode strings and non-Unicode strings slightly differently. If a string has been identified as being UTF-8 encoded, Perl will set a flag in the SV, `SVf_UTF8`. You can check and manipulate this flag with the following macros:

```
SvUTF8(sv)
SvUTF8_on(sv)
SvUTF8_off(sv)
```

This flag has an important effect on Perl's treatment of the string: if Unicode data is not properly distinguished, regular expressions, `length`, `substr` and other string handling operations will have undesirable results.

The problem comes when you have, for instance, a string that isn't flagged as UTF8, and contains a byte sequence that could be UTF8 – especially when combining non-UTF8 and UTF8 strings.

Never forget that the `SVf_UTF8` flag is separate to the PV value; you need be sure you don't accidentally knock it off while you're manipulating SVs. More specifically, you cannot expect to do this:

```
SV *sv;
SV *nsv;
STRLEN len;
```

```
char *p;

p = SvPV(sv, len);
froblicate(p);
nsv = newSVpv(p, len);
```

The `char*` string does not tell you the whole story, and you can't copy or reconstruct an SV just by copying the string value. Check if the old SV has the UTF8 flag set, and act accordingly:

```
p = SvPV(sv, len);
froblicate(p);
nsv = newSVpv(p, len);
if (SvUTF8(sv))
    SvUTF8_on(nsv);
```

In fact, your `froblicate` function should be made aware of whether or not it's dealing with UTF8 data, so that it can handle the string appropriately.

How do I convert a string to UTF8?

If you're mixing UTF8 and non-UTF8 strings, you might find it necessary to upgrade one of the strings to UTF8. If you've got an SV, the easiest way to do this is:

```
sv_utf8_upgrade(sv);
```

However, you must not do this, for example:

```
if (!SvUTF8(left))
    sv_utf8_upgrade(left);
```

If you do this in a binary operator, you will actually change one of the strings that came into the operator, and, while it shouldn't be noticeable by the end user, it can cause problems.

Instead, `bytes_to_utf8` will give you a UTF8-encoded **copy** of its string argument. This is useful for having the data available for comparisons and so on, without harming the original SV. There's also `utf8_to_bytes` to go the other way, but naturally, this will fail if the string contains any characters above 255 that can't be represented in a single byte.

Is there anything else I need to know?

Not really. Just remember these things:

- There's no way to tell if a string is UTF8 or not. You can tell if an SV is UTF8 by looking at its `SvUTF8` flag. Don't forget to set the flag if something should be UTF8. Treat the flag as part of the PV, even though it's not – if you pass on the PV to somewhere, pass on the flag too.
- If a string is UTF8, **always** use `utf8_to_uv` to get at the value, unless `!(*s & 0x80)` in which case you can use `*s`.
- When writing to a UTF8 string, **always** use `uv_to_utf8`, unless `uv < 0x80` in which case you can use `*s = uv`.
- Mixing UTF8 and non-UTF8 strings is tricky. Use `bytes_to_utf8` to get a new string which is UTF8 encoded. There are tricks you can use to delay deciding whether you need to use a UTF8 string until you get to a high character – `HALF_UPGRADE` is one of those.

Custom Operators

Custom operator support is a new experimental feature that allows you to define your own ops. This is primarily to allow the building of interpreters for other languages in the Perl core, but it also allows optimizations through the creation of "macro-ops" (ops which perform the functions of multiple ops which are usually executed together, such as `gvsv`, `gvsv`, `add`.)

This feature is implemented as a new op type, `OP_CUSTOM`. The Perl core does not "know" anything special about this op type, and so it will not be involved in any optimizations. This also means that you can define

your custom ops to be any op structure – unary, binary, list and so on – you like.

It's important to know what custom operators won't do for you. They won't let you add new syntax to Perl, directly. They won't even let you add new keywords, directly. In fact, they won't change the way Perl compiles a program at all. You have to do those changes yourself, after Perl has compiled the program. You do this either by manipulating the op tree using a CHECK block and the `B::Generate` module, or by adding a custom peephole optimizer with the `optimize` module.

When you do this, you replace ordinary Perl ops with custom ops by creating ops with the type `OP_CUSTOM` and the `pp_addr` of your own PP function. This should be defined in XS code, and should look like the PP ops in `pp_*.c`. You are responsible for ensuring that your op takes the appropriate number of values from the stack, and you are responsible for adding stack marks if necessary.

You should also "register" your op with the Perl interpreter so that it can produce sensible error and warning messages. Since it is possible to have multiple custom ops within the one "logical" op type `OP_CUSTOM`, Perl uses the value of `o-op_ppaddr` as a key into the `PL_custom_op_descs` and `PL_custom_op_names` hashes. This means you need to enter a name and description for your op at the appropriate place in the `PL_custom_op_names` and `PL_custom_op_descs` hashes.

Forthcoming versions of `B::Generate` (version 1.0 and above) should directly support the creation of custom ops by name; `Opcodes::Custom` will provide functions which make it trivial to "register" custom ops to the Perl interpreter.

AUTHORS

Until May 1997, this document was maintained by Jeff Okamoto <okamoto@corp.hp.com>. It is now maintained as part of Perl itself by the Perl 5 Porters <perl5-porters@perl.org>.

With lots of help and suggestions from Dean Roehrich, Malcolm Beattie, Andreas Koenig, Paul Hudson, Ilya Zakharevich, Paul Marquess, Neil Bowers, Matthew Green, Tim Bunce, Spider Boardman, Ulrich Pfeifer, Stephen McCamant, and Gurusamy Sarathy.

API Listing originally by Dean Roehrich <roehrich@cray.com>.

Modifications to autogenerate the API listing (*perlapi*) by Benjamin Stuhl.

SEE ALSO

`perlapi(1)`, `perlintern(1)`, `perlxs(1)`, `perlembed(1)`

NAME

perlhack – How to hack at the Perl internals

DESCRIPTION

This document attempts to explain how Perl development takes place, and ends with some suggestions for people wanting to become bona fide porters.

The perl5-porters mailing list is where the Perl standard distribution is maintained and developed. The list can get anywhere from 10 to 150 messages a day, depending on the heatedness of the debate. Most days there are two or three patches, extensions, features, or bugs being discussed at a time.

A searchable archive of the list is at either:

<http://www.xray.mpe.mpg.de/mailling-lists/perl5-porters/>

or

<http://archive.develooper.com/perl5-porters@perl.org/>

List subscribers (the porters themselves) come in several flavours. Some are quiet curious lurkers, who rarely pitch in and instead watch the ongoing development to ensure they're forewarned of new changes or features in Perl. Some are representatives of vendors, who are there to make sure that Perl continues to compile and work on their platforms. Some patch any reported bug that they know how to fix, some are actively patching their pet area (threads, Win32, the regexp engine), while others seem to do nothing but complain. In other words, it's your usual mix of technical people.

Over this group of porters presides Larry Wall. He has the final word in what does and does not change in the Perl language. Various releases of Perl are shepherded by a "pumpking", a porter responsible for gathering patches, deciding on a patch-by-patch feature-by-feature basis what will and will not go into the release. For instance, Gurusamy Sarathy was the pumpking for the 5.6 release of Perl, and Jarkko Hietaniemi is the pumpking for the 5.8 release, and Hugo van der Sanden will be the pumpking for the 5.10 release.

In addition, various people are pumpkings for different things. For instance, Andy Dougherty and Jarkko Hietaniemi share the *Configure* pumpkin.

Larry sees Perl development along the lines of the US government: there's the Legislature (the porters), the Executive branch (the pumpkings), and the Supreme Court (Larry). The legislature can discuss and submit patches to the executive branch all they like, but the executive branch is free to veto them. Rarely, the Supreme Court will side with the executive branch over the legislature, or the legislature over the executive branch. Mostly, however, the legislature and the executive branch are supposed to get along and work out their differences without impeachment or court cases.

You might sometimes see reference to Rule 1 and Rule 2. Larry's power as Supreme Court is expressed in The Rules:

- 1 Larry is always by definition right about how Perl should behave. This means he has final veto power on the core functionality.
- 2 Larry is allowed to change his mind about any matter at a later date, regardless of whether he previously invoked Rule 1.

Got that? Larry is always right, even when he was wrong. It's rare to see either Rule exercised, but they are often alluded to.

New features and extensions to the language are contentious, because the criteria used by the pumpkings, Larry, and other porters to decide which features should be implemented and incorporated are not codified in a few small design goals as with some other languages. Instead, the heuristics are flexible and often difficult to fathom. Here is one person's list, roughly in decreasing order of importance, of heuristics that new features have to be weighed against:

Does concept match the general goals of Perl?

These haven't been written anywhere in stone, but one approximation is:

1. Keep it fast, simple, and useful.
2. Keep features/concepts as orthogonal as possible.
3. No arbitrary limits (platforms, data sizes, cultures).
4. Keep it open and exciting to use/patch/advocate Perl everywhere.
5. Either assimilate new technologies, or build bridges to them.

Where is the implementation?

All the talk in the world is useless without an implementation. In almost every case, the person or people who argue for a new feature will be expected to be the ones who implement it. Porters capable of coding new features have their own agendas, and are not available to implement your (possibly good) idea.

Backwards compatibility

It's a cardinal sin to break existing Perl programs. New warnings are contentious—some say that a program that emits warnings is not broken, while others say it is. Adding keywords has the potential to break programs, changing the meaning of existing token sequences or functions might break programs.

Could it be a module instead?

Perl 5 has extension mechanisms, modules and XS, specifically to avoid the need to keep changing the Perl interpreter. You can write modules that export functions, you can give those functions prototypes so they can be called like built-in functions, you can even write XS code to mess with the runtime data structures of the Perl interpreter if you want to implement really complicated things. If it can be done in a module instead of in the core, it's highly unlikely to be added.

Is the feature generic enough?

Is this something that only the submitter wants added to the language, or would it be broadly useful? Sometimes, instead of adding a feature with a tight focus, the porters might decide to wait until someone implements the more generalized feature. For instance, instead of implementing a "delayed evaluation" feature, the porters are waiting for a macro system that would permit delayed evaluation and much more.

Does it potentially introduce new bugs?

Radical rewrites of large chunks of the Perl interpreter have the potential to introduce new bugs. The smaller and more localized the change, the better.

Does it preclude other desirable features?

A patch is likely to be rejected if it closes off future avenues of development. For instance, a patch that placed a true and final interpretation on prototypes is likely to be rejected because there are still options for the future of prototypes that haven't been addressed.

Is the implementation robust?

Good patches (tight code, complete, correct) stand more chance of going in. Sloppy or incorrect patches might be placed on the back burner until the pumpking has time to fix, or might be discarded altogether without further notice.

Is the implementation generic enough to be portable?

The worst patches make use of a system-specific features. It's highly unlikely that nonportable additions to the Perl language will be accepted.

Is the implementation tested?

Patches which change behaviour (fixing bugs or introducing new features) must include regression tests to verify that everything works as expected. Without tests provided by the original author, how can anyone else changing perl in the future be sure that they haven't unwittingly broken the behaviour

the patch implements? And without tests, how can the patch's author be confident that his/her hard work put into the patch won't be accidentally thrown away by someone in the future?

Is there enough documentation?

Patches without documentation are probably ill-thought out or incomplete. Nothing can be added without documentation, so submitting a patch for the appropriate manpages as well as the source code is always a good idea.

Is there another way to do it?

Larry said "Although the Perl Slogan is *There's More Than One Way to Do It*, I hesitate to make 10 ways to do something". This is a tricky heuristic to navigate, though—one man's essential addition is another man's pointless cruft.

Does it create too much work?

Work for the pumpking, work for Perl programmers, work for module authors, ... Perl is supposed to be easy.

Patches speak louder than words

Working code is always preferred to pie-in-the-sky ideas. A patch to add a feature stands a much higher chance of making it to the language than does a random feature request, no matter how fervently argued the request might be. This ties into "Will it be useful?", as the fact that someone took the time to make the patch demonstrates a strong desire for the feature.

If you're on the list, you might hear the word "core" bandied around. It refers to the standard distribution. "Hacking on the core" means you're changing the C source code to the Perl interpreter. "A core module" is one that ships with Perl.

Keeping in sync

The source code to the Perl interpreter, in its different versions, is kept in a repository managed by a revision control system (which is currently the Perforce program, see <http://perforce.com/>). The pumpkings and a few others have access to the repository to check in changes. Periodically the pumpking for the development version of Perl will release a new version, so the rest of the porters can see what's changed. The current state of the main trunk of repository, and patches that describe the individual changes that have happened since the last public release are available at this location:

```
http://public.activestate.com/gsar/APC/  
ftp://ftp.linux.activestate.com/pub/staff/gsar/APC/
```

If you're looking for a particular change, or a change that affected a particular set of files, you may find the **Perl Repository Browser** useful:

```
http://public.activestate.com/cgi-bin/perlbrowse
```

You may also want to subscribe to the perl5-changes mailing list to receive a copy of each patch that gets submitted to the maintenance and development "branches" of the perl repository. See <http://lists.perl.org/> for subscription information.

If you are a member of the perl5-porters mailing list, it is a good thing to keep in touch with the most recent changes. If not only to verify if what you would have posted as a bug report isn't already solved in the most recent available perl development branch, also known as perl-current, bleeding edge perl, bleedperl or bleedperl.

Needless to say, the source code in perl-current is usually in a perpetual state of evolution. You should expect it to be very buggy. Do **not** use it for any purpose other than testing and development.

Keeping in sync with the most recent branch can be done in several ways, but the most convenient and reliable way is using **rsync**, available at <ftp://rsync.samba.org/pub/rsync/> . (You can also get the most recent branch by FTP.)

If you choose to keep in sync using rsync, there are two approaches to doing so:

rsync'ing the source tree

Presuming you are in the directory where your perl source resides and you have rsync installed and available, you can 'upgrade' to the bleadperl using:

```
# rsync -avz rsync://ftp.linux.activestate.com/perl-current/ .
```

This takes care of updating every single item in the source tree to the latest applied patch level, creating files that are new (to your distribution) and setting date/time stamps of existing files to reflect the bleadperl status.

Note that this will not delete any files that were in '.' before the rsync. Once you are sure that the rsync is running correctly, run it with the `--delete` and the `--dry-run` options like this:

```
# rsync -avz --delete --dry-run rsync://ftp.linux.activestate.com/perl-current/
```

This will *simulate* an rsync run that also deletes files not present in the bleadperl master copy. Observe the results from this run closely. If you are sure that the actual run would delete no files precious to you, you could remove the `--dry-run` option.

You can then check what patch was the latest that was applied by looking in the file `.patch`, which will show the number of the latest patch.

If you have more than one machine to keep in sync, and not all of them have access to the WAN (so you are not able to rsync all the source trees to the real source), there are some ways to get around this problem.

Using rsync over the LAN

Set up a local rsync server which makes the rsynced source tree available to the LAN and sync the other machines against this directory.

From <http://rsync.samba.org/README.html> :

```
"Rsync uses rsh or ssh for communication. It does not need to be
setuid and requires no special privileges for installation. It
does not require an inetd entry or a daemon. You must, however,
have a working rsh or ssh system. Using ssh is recommended for
its security features."
```

Using pushing over the NFS

Having the other systems mounted over the NFS, you can take an active pushing approach by checking the just updated tree against the other not-yet synced trees. An example would be

```
#!/usr/bin/perl -w

use strict;
use File::Copy;

my %MF = map {
    m/(\S+)/;
    $_ => [ (stat $_)[2, 7, 9] ];      # mode, size, mtime
} 'cat MANIFEST';

my %remote = map { $_ => "/$_/pro/3gl/CPAN/perl-5.7.1" } qw(host1 host2);

foreach my $host (keys %remote) {
    unless (-d $remote{$host}) {
        print STDERR "Cannot Xsync for host $host\n";
        next;
    }
    foreach my $file (keys %MF) {
        my $rfile = "$remote{$host}/$file";
```

```

my ($mode, $size, $mtime) = (stat $rfile)[2, 7, 9];
defined $size or ($mode, $size, $mtime) = (0, 0, 0);
$size == $MF{$file}[1] && $mtime == $MF{$file}[2] and next;
printf "%4s %-34s %8d %9d %8d %9d\n",
    $host, $file, $MF{$file}[1], $MF{$file}[2], $size, $mtime;
unlink $rfile;
copy ($file, $rfile);
utime time, $MF{$file}[2], $rfile;
chmod $MF{$file}[0], $rfile;
}
}

```

though this is not perfect. It could be improved with checking file checksums before updating. Not all NFS systems support reliable utime support (when used over the NFS).

rsync'ing the patches

The source tree is maintained by the pumpking who applies patches to the files in the tree. These patches are either created by the pumpking himself using `diff -c` after updating the file manually or by applying patches sent in by posters on the perl5-porters list. These patches are also saved and rsync'able, so you can apply them yourself to the source files.

Presuming you are in a directory where your patches reside, you can get them in sync with

```
# rsync -avz rsync://ftp.linux.activestate.com/perl-current-diffs/ .
```

This makes sure the latest available patch is downloaded to your patch directory.

It's then up to you to apply these patches, using something like

```
# last=`ls -t *.gz | sed q`
# rsync -avz rsync://ftp.linux.activestate.com/perl-current-diffs/ .
# find . -name '*.gz' -newer $last -exec gzcat {} \; >blead.patch
# cd ../perl-current
# patch -p1 -N <../perl-current-diffs/blead.patch

```

or, since this is only a hint towards how it works, use CPAN-patchperl from Andreas König to have better control over the patching process.

Why rsync the source tree

It's easier to rsync the source tree

Since you don't have to apply the patches yourself, you are sure all files in the source tree are in the right state.

It's more reliable

While both the rsync-able source and patch areas are automatically updated every few minutes, keep in mind that applying patches may sometimes mean careful hand-holding, especially if your version of the patch program does not understand how to deal with new files, files with 8-bit characters, or files without trailing newlines.

Why rsync the patches

It's easier to rsync the patches

If you have more than one machine that you want to keep in track with bleadperl, it's easier to rsync the patches only once and then apply them to all the source trees on the different machines.

In case you try to keep in pace on 5 different machines, for which only one of them has access to the WAN, rsync'ing all the source trees should than be done 5 times over the NFS. Having rsync'ed the patches only once, I can apply them to all the source trees automatically. Need you say more ;-)

It's a good reference

If you do not only like to have the most recent development branch, but also like to **fix** bugs, or extend features, you want to dive into the sources. If you are a seasoned perl core diver, you don't need no manuals, tips, roadmaps, perlguys.pod or other aids to find your way around. But if you are a starter, the patches may help you in finding where you should start and how to change the bits that bug you.

The file **Changes** is updated on occasions the pumpking sees as his own little sync points. On those occasions, he releases a tar-ball of the current source tree (i.e. perl@7582.tar.gz), which will be an excellent point to start with when choosing to use the 'rsync the patches' scheme. Starting with perl@7582, which means a set of source files on which the latest applied patch is number 7582, you apply all succeeding patches available from then on (7583, 7584, ...).

You can use the patches later as a kind of search archive.

Finding a start point

If you want to fix/change the behaviour of function/feature Foo, just scan the patches for patches that mention Foo either in the subject, the comments, or the body of the fix. A good chance the patch shows you the files that are affected by that patch which are very likely to be the starting point of your journey into the guts of perl.

Finding how to fix a bug

If you've found *where* the function/feature Foo misbehaves, but you don't know how to fix it (but you do know the change you want to make), you can, again, peruse the patches for similar changes and look how others apply the fix.

Finding the source of misbehaviour

When you keep in sync with bleadperl, the pumpking would love to *see* that the community efforts really work. So after each of his sync points, you are to 'make test' to check if everything is still in working order. If it is, you do 'make ok', which will send an OK report to perlbug@perl.org. (If you do not have access to a mailer from the system you just finished successfully 'make test', you can do 'make okfile', which creates the file perl.ok, which you can then take to your favourite mailer and mail yourself).

But of course, as always, things will not always lead to a success path, and one or more test do not pass the 'make test'. Before sending in a bug report (using 'make nok' or 'make nokfile'), check the mailing list if someone else has reported the bug already and if so, confirm it by replying to that message. If not, you might want to trace the source of that misbehaviour **before** sending in the bug, which will help all the other porters in finding the solution.

Here the saved patches come in very handy. You can check the list of patches to see which patch changed what file and what change caused the misbehaviour. If you note that in the bug report, it saves the one trying to solve it, looking for that point.

If searching the patches is too bothersome, you might consider using perl's bugtron to find more information about discussions and ramblings on posted bugs.

If you want to get the best of both worlds, rsync both the source tree for convenience, reliability and ease and rsync the patches for reference.

Perlbug remote interface

There are three (3) remote administrative interfaces for modifying bug status, category, etc. In all cases an admin must be first registered with the Perlbug database by sending an email request to richard@perl.org or bugmongers@perl.org.

The main requirement is the willingness to classify, (with the emphasis on closing where possible :), outstanding bugs. Further explanation can be garnered from the web at <http://bugs.perl.org/>, or by asking on the admin mailing list at: bugmongers@perl.org

For more info on the web see

`http://bugs.perl.org/perlbug.cgi?req=spec`

1 `http://bugs.perl.org`

Login via the web, (remove **admin/** if only browsing), where interested Cc's, tests, patches and change-ids, etc. may be assigned.

`http://bugs.perl.org/admin/index.html`

2 `bugdb@perl.org`

Where the subject line is used for commands:

To: `bugdb@perl.org`
Subject: `-a close bugid1 bugid2 aix install`

To: `bugdb@perl.org`
Subject: `-h`

3 `commands_and_bugids@bugs.perl.org`

Where the address itself is the source for the commands:

To: `close_bugid1_bugid2_aix@bugs.perl.org`
To: `help@bugs.perl.org`

notes, patches, tests

For patches and tests, the message body is assigned to the appropriate bugs and forwarded to p5p for their attention.

To: `test_<bugid1>_aix_close@bugs.perl.org`
Subject: `this is a test for the (now closed) aix bug`

Test is the body of the mail

Submitting patches

Always submit patches to `perl5-porters@perl.org`. If you're patching a core module and there's an author listed, send the author a copy (see [Patching a core module](#)). This lets other porters review your patch, which catches a surprising number of errors in patches. Either use the diff program (available in source code form from `ftp://ftp.gnu.org/pub/gnu/`, or use Johan Vromans' *makepatch* (available from `CPAN/authors/id/JV/`). Unified diffs are preferred, but context diffs are accepted. Do not send RCS-style diffs or diffs without context lines. More information is given in the *Porting/patching.pod* file in the Perl source distribution. Please patch against the latest **development** version (e.g., if you're fixing a bug in the 5.005 track, patch against the latest 5.005_5x version). Only patches that survive the heat of the development branch get applied to maintenance versions.

Your patch should update the documentation and test suite. See [Writing a test](#).

To report a bug in Perl, use the program *perlbug* which comes with Perl (if you can't get Perl to work, send mail to the address `perlbug@perl.org` or `perlbug@perl.com`). Reporting bugs through *perlbug* feeds into the automated bug-tracking system, access to which is provided through the web at `http://bugs.perl.org/`. It often pays to check the archives of the perl5-porters mailing list to see whether the bug you're reporting has been reported before, and if so whether it was considered a bug. See above for the location of the searchable archives.

The CPAN testers (`http://testers.cpan.org/`) are a group of volunteers who test CPAN modules on a variety of platforms. Perl Smokers (`http://archives.develooper.com/daily-build@perl.org/`) automatically tests Perl source releases on platforms with various configurations. Both efforts welcome volunteers.

It's a good idea to read and lurk for a while before chipping in. That way you'll get to see the dynamic of the conversations, learn the personalities of the players, and hopefully be better prepared to make a useful contribution when do you speak up.

If after all this you still think you want to join the perl5-porters mailing list, send mail to

perl5-porters-subscribe@perl.org. To unsubscribe, send mail to *perl5-porters-unsubscribe@perl.org*.

To hack on the Perl guts, you'll need to read the following things:

perlguts

This is of paramount importance, since it's the documentation of what goes where in the Perl source. Read it over a couple of times and it might start to make sense – don't worry if it doesn't yet, because the best way to study it is to read it in conjunction with poking at Perl source, and we'll do that later on.

You might also want to look at Gisle Aas's illustrated *perlguts* – there's no guarantee that this will be absolutely up-to-date with the latest documentation in the Perl core, but the fundamentals will be right. (<http://gisle.aas.no/perl/illguts/>)

perlxs and *perlxs*

A working knowledge of XSUB programming is incredibly useful for core hacking; XSUBs use techniques drawn from the PP code, the portion of the guts that actually executes a Perl program. It's a lot gentler to learn those techniques from simple examples and explanation than from the core itself.

perlapi

The documentation for the Perl API explains what some of the internal functions do, as well as the many macros used in the source.

Porting/pumpkin.pod

This is a collection of words of wisdom for a Perl porter; some of it is only useful to the pumpkin holder, but most of it applies to anyone wanting to go about Perl development.

The perl5-porters FAQ

This should be available from <http://simon-cozens.org/writings/p5p-faq> ; alternatively, you can get the FAQ emailed to you by sending mail to *perl5-porters-faq@perl.org*. It contains hints on reading perl5-porters, information on how perl5-porters works and how Perl development in general works.

Finding Your Way Around

Perl maintenance can be split into a number of areas, and certain people (pumpkins) will have responsibility for each area. These areas sometimes correspond to files or directories in the source kit. Among the areas are:

Core modules

Modules shipped as part of the Perl core live in the *lib/* and *ext/* subdirectories: *lib/* is for the pure-Perl modules, and *ext/* contains the core XS modules.

Tests

There are tests for nearly all the modules, built-ins and major bits of functionality. Test files all have a .t suffix. Module tests live in the *lib/* and *ext/* directories next to the module being tested. Others live in *t/*. See [Writing a test](#)

Documentation

Documentation maintenance includes looking after everything in the *pod/* directory, (as well as contributing new documentation) and the documentation to the modules in core.

Configure

The configure process is the way we make Perl portable across the myriad of operating systems it supports. Responsibility for the configure, build and installation process, as well as the overall portability of the core code rests with the configure pumpkin – others help out with individual operating systems.

The files involved are the operating system directories, (*win32/*, *os2/*, *vms/* and so on) the shell scripts which generate *config.h* and *Makefile*, as well as the metaconfig files which generate *Configure*. (metaconfig isn't included in the core distribution.)

Interpreter

And of course, there's the core of the Perl interpreter itself. Let's have a look at that in a little more detail.

Before we leave looking at the layout, though, don't forget that *MANIFEST* contains not only the file names in the Perl distribution, but short descriptions of what's in them, too. For an overview of the important files, try this:

```
perl -lne 'print if /^[\^\/]+\.[ch]\s+/' MANIFEST
```

Elements of the interpreter

The work of the interpreter has two main stages: compiling the code into the internal representation, or bytecode, and then executing it. *Compiled code* explains exactly how the compilation stage happens.

Here is a short breakdown of perl's operation:

Startup

The action begins in *perlmain.c*. (or *miniperlmain.c* for miniperl) This is very high-level code, enough to fit on a single screen, and it resembles the code found in *perlembed*; most of the real action takes place in *perl.c*

First, *perlmain.c* allocates some memory and constructs a Perl interpreter:

```
1 PERL_SYS_INIT3 (&argc, &argv, &env) ;
2
3 if (!PL_do_undump) {
4     my_perl = perl_alloc();
5     if (!my_perl)
6         exit(1);
7     perl_construct(my_perl);
8     PL_perl_destruct_level = 0;
9 }
```

Line 1 is a macro, and its definition is dependent on your operating system. Line 3 references `PL_do_undump`, a global variable – all global variables in Perl start with `PL_`. This tells you whether the current running program was created with the `-u` flag to perl and then *undump*, which means it's going to be false in any sane context.

Line 4 calls a function in *perl.c* to allocate memory for a Perl interpreter. It's quite a simple function, and the guts of it looks like this:

```
my_perl = (PerlInterpreter*) PerlMem_malloc(sizeof(PerlInterpreter));
```

Here you see an example of Perl's system abstraction, which we'll see later: `PerlMem_malloc` is either your system's `malloc`, or Perl's own `malloc` as defined in *malloc.c* if you selected that option at configure time.

Next, in line 7, we construct the interpreter; this sets up all the special variables that Perl needs, the stacks, and so on.

Now we pass Perl the command line options, and tell it to go:

```
exitstatus = perl_parse(my_perl, xs_init, argc, argv, (char **)NULL);
if (!exitstatus) {
    exitstatus = perl_run(my_perl);
}
```

`perl_parse` is actually a wrapper around `S_parse_body`, as defined in *perl.c*, which processes the command line options, sets up any statically linked XS modules, opens the program and calls `yyparse` to parse it.

Parsing

The aim of this stage is to take the Perl source, and turn it into an op tree. We'll see what one of those looks like later. Strictly speaking, there's three things going on here.

`yyparse`, the parser, lives in *perly.c*, although you're better off reading the original YACC input in *perly.y*. (Yes, Virginia, there **is** a YACC grammar for Perl!) The job of the parser is to take your code and 'understand' it, splitting it into sentences, deciding which operands go with which operators and so on.

The parser is nobly assisted by the lexer, which chunks up your input into tokens, and decides what type of thing each token is: a variable name, an operator, a bareword, a subroutine, a core function, and so on. The main point of entry to the lexer is `yylex`, and that and its associated routines can be found in *token.c*. Perl isn't much like other computer languages; it's highly context sensitive at times, it can be tricky to work out what sort of token something is, or where a token ends. As such, there's a lot of interplay between the tokeniser and the parser, which can get pretty frightening if you're not used to it.

As the parser understands a Perl program, it builds up a tree of operations for the interpreter to perform during execution. The routines which construct and link together the various operations are to be found in *op.c*, and will be examined later.

Optimization

Now the parsing stage is complete, and the finished tree represents the operations that the Perl interpreter needs to perform to execute our program. Next, Perl does a dry run over the tree looking for optimisations: constant expressions such as `3 + 4` will be computed now, and the optimizer will also see if any multiple operations can be replaced with a single one. For instance, to fetch the variable `$foo`, instead of grabbing the glob `*foo` and looking at the scalar component, the optimizer fiddles the op tree to use a function which directly looks up the scalar in question. The main optimizer is `peep` in *op.c*, and many ops have their own optimizing functions.

Running

Now we're finally ready to go: we have compiled Perl byte code, and all that's left to do is run it. The actual execution is done by the `runops_standard` function in *run.c*; more specifically, it's done by these three innocent looking lines:

```
while ((PL_op = CALL_FPTR(PL_op->op_ppaddr) (aTHX))) {
    PERL_ASYNC_CHECK();
}
```

You may be more comfortable with the Perl version of that:

```
PERL_ASYNC_CHECK() while $Perl::op = &{$Perl::op->{function}};
```

Well, maybe not. Anyway, each op contains a function pointer, which stipulates the function which will actually carry out the operation. This function will return the next op in the sequence – this allows for things like `if` which choose the next op dynamically at run time. The `PERL_ASYNC_CHECK` makes sure that things like signals interrupt execution if required.

The actual functions called are known as PP code, and they're spread between four files: *pp_hot.c* contains the 'hot' code, which is most often used and highly optimized, *pp_sys.c* contains all the system-specific functions, *pp_ctl.c* contains the functions which implement control structures (`if`, `while` and the like) and *pp.c* contains everything else. These are, if you like, the C code for Perl's built-in functions and operators.

Internal Variable Types

You should by now have had a look at *perlvars*, which tells you about Perl's internal variable types: SVs, HVs, AVs and the rest. If not, do that now.

These variables are used not only to represent Perl-space variables, but also any constants in the code, as well as some structures completely internal to Perl. The symbol table, for instance, is an ordinary Perl hash.

Your code is represented by an SV as it's read into the parser; any program files you call are opened via ordinary Perl filehandles, and so on.

The core [Devel::Peek/Devel::Peek](#) module lets us examine SVs from a Perl program. Let's see, for instance, how Perl treats the constant "hello".

```
% perl -MDevel::Peek -e 'Dump("hello")'
1 SV = PV(0xa041450) at 0xa04ecbc
2   REFCNT = 1
3   FLAGS = (POK,READONLY,pPOK)
4   PV = 0xa0484e0 "hello"\0
5   CUR = 5
6   LEN = 6
```

Reading `Devel::Peek` output takes a bit of practise, so let's go through it line by line.

Line 1 tells us we're looking at an SV which lives at 0xa04ecbc in memory. SVs themselves are very simple structures, but they contain a pointer to a more complex structure. In this case, it's a PV, a structure which holds a string value, at location 0xa041450. Line 2 is the reference count; there are no other references to this data, so it's 1.

Line 3 are the flags for this SV – it's OK to use it as a PV, it's a read-only SV (because it's a constant) and the data is a PV internally. Next we've got the contents of the string, starting at location 0xa0484e0.

Line 5 gives us the current length of the string – note that this does **not** include the null terminator. Line 6 is not the length of the string, but the length of the currently allocated buffer; as the string grows, Perl automatically extends the available storage via a routine called `SvGROW`.

You can get at any of these quantities from C very easily; just add `Sv` to the name of the field shown in the snippet, and you've got a macro which will return the value: `SvCUR(sv)` returns the current length of the string, `SvREFCOUNT(sv)` returns the reference count, `SvPV(sv, len)` returns the string itself with its length, and so on. More macros to manipulate these properties can be found in [perlguts](#).

Let's take an example of manipulating a PV, from `sv_catpv`, in `sv.c`

```
1 void
2 Perl_sv_catpv(pTHX_ register SV *sv, register const char *ptr, register STRLEN
3 {
4     STRLEN tlen;
5     char *junk;
6
7     junk = SvPV_force(sv, tlen);
8     SvGROW(sv, tlen + len + 1);
9     if (ptr == junk)
10         ptr = SvPVX(sv);
11     Move(ptr, SvPVX(sv)+tlen, len, char);
12     SvCUR(sv) += len;
13     *SvEND(sv) = '\0';
14     (void)SvPOK_only_UTF8(sv);          /* validate pointer */
15     SvTAINT(sv);
16 }
```

This is a function which adds a string, `ptr`, of length `len` onto the end of the PV stored in `sv`. The first thing we do in line 6 is make sure that the SV **has** a valid PV, by calling the `SvPV_force` macro to force a PV. As a side effect, `tlen` gets set to the current value of the PV, and the PV itself is returned to `junk`.

In line 7, we make sure that the SV will have enough room to accommodate the old string, the new string and the null terminator. If `LEN` isn't big enough, `SvGROW` will reallocate space for us.

Now, if `junk` is the same as the string we're trying to add, we can grab the string directly from the SV; `SvPVX` is the address of the PV in the SV.

Line 10 does the actual catenation: the `Move` macro moves a chunk of memory around: we move the string `ptr` to the end of the `PV` – that's the start of the `PV` plus its current length. We're moving `len` bytes of type `char`. After doing so, we need to tell Perl we've extended the string, by altering `CUR` to reflect the new length. `SvEND` is a macro which gives us the end of the string, so that needs to be a `"\0"`.

Line 13 manipulates the flags; since we've changed the `PV`, any `IV` or `NV` values will no longer be valid: if we have `$a=10; $a.="6";` we don't want to use the old `IV` of 10. `SvPOK_only_utf8` is a special UTF8-aware version of `SvPOK_only`, a macro which turns off the `IOK` and `NOK` flags and turns on `POK`. The final `SvTAINT` is a macro which launders tainted data if taint mode is turned on.

`AVs` and `HVs` are more complicated, but `SVs` are by far the most common variable type being thrown around. Having seen something of how we manipulate these, let's go on and look at how the op tree is constructed.

Op Trees

First, what is the op tree, anyway? The op tree is the parsed representation of your program, as we saw in our section on parsing, and it's the sequence of operations that Perl goes through to execute your program, as we saw in [/Running](#).

An op is a fundamental operation that Perl can perform: all the built-in functions and operators are ops, and there are a series of ops which deal with concepts the interpreter needs internally – entering and leaving a block, ending a statement, fetching a variable, and so on.

The op tree is connected in two ways: you can imagine that there are two "routes" through it, two orders in which you can traverse the tree. First, parse order reflects how the parser understood the code, and secondly, execution order tells perl what order to perform the operations in.

The easiest way to examine the op tree is to stop Perl after it has finished parsing, and get it to dump out the tree. This is exactly what the compiler backends [B::Terse/B::Terse](#), [B::Concise/B::Concise](#) and [B::Debug/B::Debug](#) do.

Let's have a look at how Perl sees `$a = $b + $c`:

```
% perl -MO=Terse -e '$a=$b+$c'
1 LISTOP (0x8179888) leave
2      OP (0x81798b0) enter
3      COP (0x8179850) nextstate
4      BINOP (0x8179828) sassign
5          BINOP (0x8179800) add [1]
6              UNOP (0x81796e0) null [15]
7                  SVOP (0x80fafe0) gvsv GV (0x80fa4cc) *b
8                      UNOP (0x81797e0) null [15]
9                          SVOP (0x8179700) gvsv GV (0x80efeb0) *c
10                  UNOP (0x816b4f0) null [15]
11                      SVOP (0x816dcf0) gvsv GV (0x80fa460) *a
```

Let's start in the middle, at line 4. This is a `BINOP`, a binary operator, which is at location `0x8179828`. The specific operator in question is `sassign` – scalar assignment – and you can find the code which implements it in the function `pp_sassign` in *pp_hot.c*. As a binary operator, it has two children: the add operator, providing the result of `$b+$c`, is uppermost on line 5, and the left hand side is on line 10.

Line 10 is the null op: this does exactly nothing. What is that doing there? If you see the null op, it's a sign that something has been optimized away after parsing. As we mentioned in [/Optimization](#), the optimization stage sometimes converts two operations into one, for example when fetching a scalar variable. When this happens, instead of rewriting the op tree and cleaning up the dangling pointers, it's easier just to replace the redundant operation with the null op. Originally, the tree would have looked like this:

```
10          SVOP (0x816b4f0) rv2sv [15]
11          SVOP (0x816dcf0) gv GV (0x80fa460) *a
```

That is, fetch the `a` entry from the main symbol table, and then look at the scalar component of it: `gvsv` (`pp_gvsv` into *pp_hot.c*) happens to do both these things.

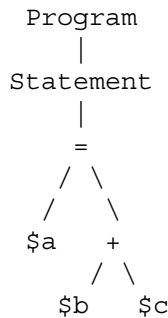
The right hand side, starting at line 5 is similar to what we've just seen: we have the `add` op (`pp_add` also in *pp_hot.c*) add together two `gvsvs`.

Now, what's this about?

```
1 LISTOP (0x8179888) leave
2      OP (0x81798b0) enter
3      COP (0x8179850) nextstate
```

`enter` and `leave` are scoping ops, and their job is to perform any housekeeping every time you enter and leave a block: lexical variables are tidied up, unreferenced variables are destroyed, and so on. Every program will have those first three lines: `leave` is a list, and its children are all the statements in the block. Statements are delimited by `nextstate`, so a block is a collection of `nextstate` ops, with the ops to be performed for each statement being the children of `nextstate`. `enter` is a single op which functions as a marker.

That's how Perl parsed the program, from top to bottom:



However, it's impossible to **perform** the operations in this order: you have to find the values of `$b` and `$c` before you add them together, for instance. So, the other thread that runs through the op tree is the execution order: each op has a field `op_next` which points to the next op to be run, so following these pointers tells us how perl executes the code. We can traverse the tree in this order using the `exec` option to `B::Terse`:

```
% perl -MO=Terse,exec -e '$a=$b+$c'
1 OP (0x8179928) enter
2 COP (0x81798c8) nextstate
3 SVOP (0x81796c8) gvsv GV (0x80fa4d4) *b
4 SVOP (0x8179798) gvsv GV (0x80efeb0) *c
5 BINOP (0x8179878) add [1]
6 SVOP (0x816dd38) gvsv GV (0x80fa468) *a
7 BINOP (0x81798a0) sassign
8 LISTOP (0x8179900) leave
```

This probably makes more sense for a human: enter a block, start a statement. Get the values of `$b` and `$c`, and add them together. Find `$a`, and assign one to the other. Then leave.

The way Perl builds up these op trees in the parsing process can be unravelled by examining *perly.y*, the YACC grammar. Let's take the piece we need to construct the tree for `$a = $b + $c`

```
1 term      :   term ASSIGNOP term
2             { $$ = newASSIGNOP(OPf_STACKED, $1, $2, $3); }
3           |   term ADDOP term
4             { $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

If you're not used to reading BNF grammars, this is how it works: You're fed certain things by the tokeniser, which generally end up in upper case. Here, `ADDOP`, is provided when the tokeniser sees `+` in your code.

ASSIGNOP is provided when = is used for assigning. These are ‘terminal symbols’, because you can’t get any simpler than them.

The grammar, lines one and three of the snippet above, tells you how to build up more complex forms. These complex forms, ‘non-terminal symbols’ are generally placed in lower case. `term` here is a non-terminal symbol, representing a single expression.

The grammar gives you the following rule: you can make the thing on the left of the colon if you see all the things on the right in sequence. This is called a “reduction”, and the aim of parsing is to completely reduce the input. There are several different ways you can perform a reduction, separated by vertical bars: so, `term` followed by = followed by `term` makes a `term`, and `term` followed by + followed by `term` can also make a `term`.

So, if you see two terms with an = or +, between them, you can turn them into a single expression. When you do this, you execute the code in the block on the next line: if you see =, you’ll do the code in line 2. If you see +, you’ll do the code in line 4. It’s this code which contributes to the op tree.

```
|   term ADDOP term
{ $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

What this does is creates a new binary op, and feeds it a number of variables. The variables refer to the tokens: \$1 is the first token in the input, \$2 the second, and so on – think regular expression backreferences. \$\$ is the op returned from this reduction. So, we call `newBINOP` to create a new binary operator. The first parameter to `newBINOP`, a function in *op.c*, is the op type. It’s an addition operator, so we want the type to be `ADDOP`. We could specify this directly, but it’s right there as the second token in the input, so we use \$2. The second parameter is the op’s flags: 0 means ‘nothing special’. Then the things to add: the left and right hand side of our expression, in scalar context.

Stacks

When perl executes something like `addop`, how does it pass on its results to the next op? The answer is, through the use of stacks. Perl has a number of stacks to store things it’s currently working on, and we’ll look at the three most important ones here.

Argument stack

Arguments are passed to PP code and returned from PP code using the argument stack, `ST`. The typical way to handle arguments is to pop them off the stack, deal with them how you wish, and then push the result back onto the stack. This is how, for instance, the cosine operator works:

```
NV value;
value = POPn;
value = Perl_cos(value);
XPUSHn(value);
```

We’ll see a more tricky example of this when we consider Perl’s macros below. `POPn` gives you the NV (floating point value) of the top SV on the stack: the \$x in `cos($x)`. Then we compute the cosine, and push the result back as an NV. The X in `XPUSHn` means that the stack should be extended if necessary – it can’t be necessary here, because we know there’s room for one more item on the stack, since we’ve just removed one! The `XPUSH*` macros at least guarantee safety.

Alternatively, you can fiddle with the stack directly: `SP` gives you the first element in your portion of the stack, and `TOP*` gives you the top SV/IV/NV/etc. on the stack. So, for instance, to do unary negation of an integer:

```
SETi(-TOPi);
```

Just set the integer value of the top stack entry to its negation.

Argument stack manipulation in the core is exactly the same as it is in XSUBS – see [perlxtut](#), [perlxs](#) and [perlguts](#) for a longer description of the macros used in stack manipulation.

Mark stack

I say ‘your portion of the stack’ above because PP code doesn’t necessarily get the whole stack to itself: if your function calls another function, you’ll only want to expose the arguments aimed for the called function, and not (necessarily) let it get at your own data. The way we do this is to have a ‘virtual’ bottom-of-stack, exposed to each function. The mark stack keeps bookmarks to locations in the argument stack usable by each function. For instance, when dealing with a tied variable, (internally, something with ‘P’ magic) Perl has to call methods for accesses to the tied variables. However, we need to separate the arguments exposed to the method to the argument exposed to the original function – the store or fetch or whatever it may be. Here’s how the tied push is implemented; see `av_push` in *av.c*:

```

1  PUSHMARK (SP) ;
2  EXTEND (SP, 2) ;
3  PUSHs (SvTIED_obj ((SV*) av, mg)) ;
4  PUSHs (val) ;
5  PUTBACK;
6  ENTER;
7  call_method ("PUSH", G_SCALAR | G_DISCARD) ;
8  LEAVE;
9  POPSTACK;
```

The lines which concern the mark stack are the first, fifth and last lines: they save away, restore and remove the current position of the argument stack.

Let’s examine the whole implementation, for practice:

```

1  PUSHMARK (SP) ;
```

Push the current state of the stack pointer onto the mark stack. This is so that when we’ve finished adding items to the argument stack, Perl knows how many things we’ve added recently.

```

2  EXTEND (SP, 2) ;
3  PUSHs (SvTIED_obj ((SV*) av, mg)) ;
4  PUSHs (val) ;
```

We’re going to add two more items onto the argument stack: when you have a tied array, the `PUSH` subroutine receives the object and the value to be pushed, and that’s exactly what we have here – the tied object, retrieved with `SvTIED_obj`, and the value, the `SV val`.

```

5  PUTBACK;
```

Next we tell Perl to make the change to the global stack pointer: `dSP` only gave us a local copy, not a reference to the global.

```

6  ENTER;
7  call_method ("PUSH", G_SCALAR | G_DISCARD) ;
8  LEAVE;
```

`ENTER` and `LEAVE` localise a block of code – they make sure that all variables are tidied up, everything that has been localised gets its previous value returned, and so on. Think of them as the `{` and `}` of a Perl block.

To actually do the magic method call, we have to call a subroutine in Perl space: `call_method` takes care of that, and it’s described in *perlcalls*. We call the `PUSH` method in scalar context, and we’re going to discard its return value.

```

9  POPSTACK;
```

Finally, we remove the value we placed on the mark stack, since we don’t need it any more.

Save stack

C doesn't have a concept of local scope, so perl provides one. We've seen that `ENTER` and `LEAVE` are used as scoping braces; the save stack implements the C equivalent of, for example:

```
{
    local $foo = 42;
    ...
}
```

See [Localising Changes](#) for how to use the save stack.

Millions of Macros

One thing you'll notice about the Perl source is that it's full of macros. Some have called the pervasive use of macros the hardest thing to understand, others find it adds to clarity. Let's take an example, the code which implements the addition operator:

```
1  PP(pp_add)
2  {
3      dSP; dTARGET; tryAMAGICbin(add,opASSIGN);
4      {
5          dPOPTOPnnrl_ul;
6          SETn( left + right );
7          RETURN;
8      }
9  }
```

Every line here (apart from the braces, of course) contains a macro. The first line sets up the function declaration as Perl expects for PP code; line 3 sets up variable declarations for the argument stack and the target, the return value of the operation. Finally, it tries to see if the addition operation is overloaded; if so, the appropriate subroutine is called.

Line 5 is another variable declaration – all variable declarations start with `d` – which pops from the top of the argument stack two NVs (hence `nn`) and puts them into the variables `right` and `left`, hence the `rl`. These are the two operands to the addition operator. Next, we call `SETn` to set the NV of the return value to the result of adding the two values. This done, we return – the `RETURN` macro makes sure that our return value is properly handled, and we pass the next operator to run back to the main run loop.

Most of these macros are explained in [perlapi](#), and some of the more important ones are explained in [perlxs](#) as well. Pay special attention to [Background and PERL_IMPLICIT_CONTEXT](#) for information on the `[pad] THX_?` macros.

Poking at Perl

To really poke around with Perl, you'll probably want to build Perl for debugging, like this:

```
./Configure -d -D optimize=-g
make
```

`-g` is a flag to the C compiler to have it produce debugging information which will allow us to step through a running program. **Configure** will also turn on the `DEBUGGING` compilation symbol which enables all the internal debugging code in Perl. There are a whole bunch of things you can debug with this: [perlrun](#) lists them all, and the best way to find out about them is to play about with them. The most useful options are probably

```
l  Context (loop) stack processing
t  Trace execution
o  Method and overloading resolution
c  String/numeric conversions
```

Some of the functionality of the debugging code can be achieved using XS modules.


```
-Dr => use re 'debug'
-Dx => use O 'Debug'
```

Using a source-level debugger

If the debugging output of `-D` doesn't help you, it's time to step through perl's execution with a source-level debugger.

- We'll use `gdb` for our examples here; the principles will apply to any debugger, but check the manual of the one you're using.

To fire up the debugger, type

```
gdb ./perl
```

You'll want to do that in your Perl source tree so the debugger can read the source code. You should see the copyright message, followed by the prompt.

```
(gdb)
```

`help` will get you into the documentation, but here are the most useful commands:

`run [args]`

Run the program with the given arguments.

`break function_name`

`break source.c:xxx`

Tells the debugger that we'll want to pause execution when we reach either the named function (but see [Internal Functions!](#)) or the given line in the named source file.

`step`

Steps through the program a line at a time.

`next`

Steps through the program a line at a time, without descending into functions.

`continue`

Run until the next breakpoint.

`finish`

Run until the end of the current function, then stop again.

`'enter'`

Just pressing Enter will do the most recent operation again – it's a blessing when stepping through miles of source code.

`print`

Execute the given C code and print its results. **WARNING:** Perl makes heavy use of macros, and ***gdb*** is not aware of macros. You'll have to substitute them yourself. So, for instance, you can't say

```
print SvPV_nolen(sv)
```

but you have to say

```
print Perl_sv_2pv_nolen(sv)
```

You may find it helpful to have a "macro dictionary", which you can produce by saying `cpp -dM perl.c | sort`. Even then, ***cpp*** won't recursively apply the macros for you.

Dumping Perl Data Structures

One way to get around this macro hell is to use the dumping functions in ***dump.c***; these work a little like an internal [Devel::Peek/Devel::Peek](#), but they also cover OPs and other structures that you can't get at from Perl. Let's take an example. We'll use the `$a = $b + $c` we used before, but give it a bit of context: `$b = "6XXXX"; $c = 2.3;`. Where's a good place to stop and poke around?

What about `pp_add`, the function we examined earlier to implement the `+` operator:

```
(gdb) break Perl_pp_add
Breakpoint 1 at 0x46249f: file pp_hot.c, line 309.
```

Notice we use `Perl_pp_add` and not `pp_add` – see [Internal Functions](#). With the breakpoint in place, we can run our program:

```
(gdb) run -e '$b = "6XXXX"; $c = 2.3; $a = $b + $c'
```

Lots of junk will go past as `gdb` reads in the relevant source files and libraries, and then:

```
Breakpoint 1, Perl_pp_add () at pp_hot.c:309
309      dSP; dATARGET; tryAMAGICbin(add,opASSIGN);
(gdb) step
311      dPOPTOPnnr1_ul;
(gdb)
```

We looked at this bit of code before, and we said that `dPOPTOPnnr1_ul` arranges for two NVs to be placed into `left` and `right` – let's slightly expand it:

```
#define dPOPTOPnnr1_ul  NV right = POPn; \
                        SV *leftsv = TOPs; \
                        NV left = USE_LEFT(leftsv) ? SvNV(leftsv) : 0.0
```

`POPn` takes the `SV` from the top of the stack and obtains its `NV` either directly (if `SvNOK` is set) or by calling the `sv_2nv` function. `TOPs` takes the next `SV` from the top of the stack – yes, `POPn` uses `TOPs` – but doesn't remove it. We then use `SvNV` to get the `NV` from `leftsv` in the same way as before – yes, `POPn` uses `SvNV`.

Since we don't have an `NV` for `$b`, we'll have to use `sv_2nv` to convert it. If we step again, we'll find ourselves there:

```
Perl_sv_2nv (sv=0xa0675d0) at sv.c:1669
1669      if (!sv)
(gdb)
```

We can now use `Perl_sv_dump` to investigate the `SV`:

```
SV = PV(0xa057cc0) at 0xa0675d0
REFCNT = 1
FLAGS = (POK,pPOK)
PV = 0xa06a510 "6XXXX"\0
CUR = 5
LEN = 6
$1 = void
```

We know we're going to get 6 from this, so let's finish the subroutine:

```
(gdb) finish
Run till exit from #0  Perl_sv_2nv (sv=0xa0675d0) at sv.c:1671
0x462669 in Perl_pp_add () at pp_hot.c:311
311      dPOPTOPnnr1_ul;
```

We can also dump out this `op`: the current `op` is always stored in `PL_op`, and we can dump it with `Perl_op_dump`. This'll give us similar output to [B::Debug/B::Debug](#).

```
{
13  TYPE = add  ==> 14
    TARG = 1
    FLAGS = (SCALAR,KIDS)
    {
```

```

        TYPE = null ==> (12)
        (was rv2sv)
        FLAGS = (SCALAR, KIDS)
        {
11         TYPE = gvsv ==> 12
            FLAGS = (SCALAR)
            GV = main::b
        }
    }

# finish this later #

```

Patching

All right, we've now had a look at how to navigate the Perl sources and some things you'll need to know when fiddling with them. Let's now get on and create a simple patch. Here's something Larry suggested: if a U is the first active format during a pack, (for example, `pack "U3C8", @stuff`) then the resulting string should be treated as UTF8 encoded.

How do we prepare to fix this up? First we locate the code in question – the `pack` happens at runtime, so it's going to be in one of the *pp* files. Sure enough, `pp_pack` is in *pp.c*. Since we're going to be altering this file, let's copy it to *pp.c~*.

[Well, it was in *pp.c* when this tutorial was written. It has now been split off with `pp_unpack` to its own file, *pp_pack.c*]

Now let's look over `pp_pack`: we take a pattern into `pat`, and then loop over the pattern, taking each format character in turn into `datum_type`. Then for each possible format character, we swallow up the other arguments in the pattern (a field width, an asterisk, and so on) and convert the next chunk input into the specified format, adding it onto the output SV `cat`.

How do we know if the U is the first format in the `pat`? Well, if we have a pointer to the start of `pat` then, if we see a U we can test whether we're still at the start of the string. So, here's where `pat` is set up:

```

STRLEN fromlen;
register char *pat = SvPVx(++MARK, fromlen);
register char *patend = pat + fromlen;
register I32 len;
I32 datumtype;
SV *fromstr;

```

We'll have another string pointer in there:

```

STRLEN fromlen;
register char *pat = SvPVx(++MARK, fromlen);
register char *patend = pat + fromlen;
+ char *patcopy;
register I32 len;
I32 datumtype;
SV *fromstr;

```

And just before we start the loop, we'll set `patcopy` to be the start of `pat`:

```

items = SP - MARK;
MARK++;
sv_setpvn(cat, "", 0);
+ patcopy = pat;
while (pat < patend) {

```

Now if we see a U which was at the start of the string, we turn on the UTF8 flag for the output SV, `cat`:

```
+ if (datumtype == 'U' && pat==patcopy+1)
+     SvUTF8_on(cat);
    if (datumtype == '#') {
        while (pat < patend && *pat != '\n')
            pat++;
```

Remember that it has to be `patcopy+1` because the first character of the string is the U which has been swallowed into `datumtype`!

Oops, we forgot one thing: what if there are spaces at the start of the pattern? `pack(" U*", @stuff)` will have U as the first active character, even though it's not the first thing in the pattern. In this case, we have to advance `patcopy` along with `pat` when we see spaces:

```
    if (isSPACE(datumtype))
        continue;
```

needs to become

```
    if (isSPACE(datumtype)) {
        patcopy++;
        continue;
    }
```

OK. That's the C part done. Now we must do two additional things before this patch is ready to go: we've changed the behaviour of Perl, and so we must document that change. We must also provide some more regression tests to make sure our patch works and doesn't create a bug somewhere else along the line.

The regression tests for each operator live in *t/op/*, and so we make a copy of *t/op/pack.t* to *t/op/pack.t~*. Now we can add our tests to the end. First, we'll test that the U does indeed create Unicode strings.

t/op/pack.t has a sensible `ok()` function, but if it didn't we could use the one from *t/test.pl*.

```
require './test.pl';
plan( tests => 159 );
```

so instead of this:

```
print 'not ' unless "1.20.300.4000" eq sprintf "%vd", pack("U*",1,20,300,4000);
print "ok $test\n"; $test++;
```

we can write the more sensible (see [Test::More](#) for a full explanation of `is()` and other testing functions).

```
is( "1.20.300.4000", sprintf "%vd", pack("U*",1,20,300,4000),
    "U* produces unicode" );
```

Now we'll test that we got that space-at-the-beginning business right:

```
is( "1.20.300.4000", sprintf "%vd", pack(" U*",1,20,300,4000),
    " with spaces at the beginning" );
```

And finally we'll test that we don't make Unicode strings if U is **not** the first active format:

```
isnt( v1.20.300.4000, sprintf "%vd", pack("C0U*",1,20,300,4000),
    "U* not first isn't unicode" );
```

Mustn't forget to change the number of tests which appears at the top, or else the automated tester will get confused. This will either look like this:

```
print "1..156\n";
```

or this:

```
plan( tests => 156 );
```

We now compile up Perl, and run it through the test suite. Our new tests pass, hooray!

Finally, the documentation. The job is never done until the paperwork is over, so let's describe the change we've just made. The relevant place is *pod/perlfunc.pod*; again, we make a copy, and then we'll insert this text in the description of pack:

```
=item *
```

If the pattern begins with a C<U>, the resulting string will be treated as Unicode-encoded. You can force UTF8 encoding on in a string with an initial C<U0>, and the bytes that follow will be interpreted as Unicode characters. If you don't want this to happen, you can begin your pattern with C<C0> (or anything else) to force Perl not to UTF8 encode your string, and then follow this with a C<U*> somewhere in your pattern.

All done. Now let's create the patch. *Porting/patching.pod* tells us that if we're making major changes, we should copy the entire directory to somewhere safe before we begin fiddling, and then do

```
diff -ruN old new > patch
```

However, we know which files we've changed, and we can simply do this:

```
diff -u pp.c~ pp.c > patch
diff -u t/op/pack.t~ t/op/pack.t >> patch
diff -u pod/perlfunc.pod~ pod/perlfunc.pod >> patch
```

We end up with a patch looking a little like this:

```
--- pp.c~ Fri Jun 02 04:34:10 2000
+++ pp.c Fri Jun 16 11:37:25 2000
@@ -4375,6 +4375,7 @@
     register I32 items;
     STRLEN fromlen;
     register char *pat = SvPVx(*++MARK, fromlen);
+   char *patcopy;
     register char *patend = pat + fromlen;
     register I32 len;
     I32 datumtype;
@@ -4405,6 +4406,7 @@
...

```

And finally, we submit it, with our rationale, to perl5-porters. Job done!

Patching a core module

This works just like patching anything else, with an extra consideration. Many core modules also live on CPAN. If this is so, patch the CPAN version instead of the core and send the patch off to the module maintainer (with a copy to p5p). This will help the module maintainer keep the CPAN version in sync with the core version without constantly scanning p5p.

Adding a new function to the core

If, as part of a patch to fix a bug, or just because you have an especially good idea, you decide to add a new function to the core, discuss your ideas on p5p well before you start work. It may be that someone else has already attempted to do what you are considering and can give lots of good advice or even provide you with bits of code that they already started (but never finished).

You have to follow all of the advice given above for patching. It is extremely important to test any addition thoroughly and add new tests to explore all boundary conditions that your new function is expected to handle. If your new function is used only by one module (e.g. `toke`), then it should probably be named `S_your_function` (for static); on the other hand, if you expect it to be accessible from other functions in Perl, you should name it `Perl_your_function`. See [Internal Functions](#) for more details.

The location of any new code is also an important consideration. Don't just create a new top level `.c` file and put your code there; you would have to make changes to `Configure` (so the `Makefile` is created properly), as

well as possibly lots of include files. This is strictly pumping business.

It is better to add your function to one of the existing top level source code files, but your choice is complicated by the nature of the Perl distribution. Only the files that are marked as compiled static are located in the perl executable. Everything else is located in the shared library (or DLL if you are running under WIN32). So, for example, if a function was only used by functions located in `toke.c`, then your code can go in `toke.c`. If, however, you want to call the function from `universal.c`, then you should put your code in another location, for example `util.c`.

In addition to writing your c-code, you will need to create an appropriate entry in `embed.pl` describing your function, then run ‘make regen_headers’ to create the entries in the numerous header files that perl needs to compile correctly. See [Internal Functions](#) for information on the various options that you can set in `embed.pl`. You will forget to do this a few (or many) times and you will get warnings during the compilation phase. Make sure that you mention this when you post your patch to P5P; the pumping needs to know this.

When you write your new code, please be conscious of existing code conventions used in the perl source files. See [perlstyle](#) for details. Although most of the guidelines discussed seem to focus on Perl code, rather than c, they all apply (except when they don’t ;). See also *Porting/patching.pod* file in the Perl source distribution for lots of details about both formatting and submitting patches of your changes.

Lastly, TEST TEST TEST TEST TEST any code before posting to p5p. Test on as many platforms as you can find. Test as many perl Configure options as you can (e.g. MULTIPLICITY). If you have profiling or memory tools, see [EXTERNAL TOOLS FOR DEBUGGING PERL](#) below for how to use them to further test your code. Remember that most of the people on P5P are doing this on their own time and don’t have the time to debug your code.

Writing a test

Every module and built-in function has an associated test file (or should...). If you add or change functionality, you have to write a test. If you fix a bug, you have to write a test so that bug never comes back. If you alter the docs, it would be nice to test what the new documentation says.

In short, if you submit a patch you probably also have to patch the tests.

For modules, the test file is right next to the module itself. *lib/strict.t* tests *lib/strict.pm*. This is a recent innovation, so there are some snags (and it would be wonderful for you to brush them out), but it basically works that way. Everything else lives in *t/*.

t/base/

Testing of the absolute basic functionality of Perl. Things like `if`, basic file reads and writes, simple regexes, etc. These are run first in the test suite and if any of them fail, something is *really* broken.

t/cmd/

These test the basic control structures, `if/else`, `while`, subroutines, etc.

t/comp/

Tests basic issues of how Perl parses and compiles itself.

t/io/

Tests for built-in IO functions, including command line arguments.

t/lib/

The old home for the module tests, you shouldn’t put anything new in here. There are still some bits and pieces hanging around in here that need to be moved. Perhaps you could move them? Thanks!

t/op/

Tests for perl’s built in functions that don’t fit into any of the other directories.

t/pod/

Tests for POD directives. There are still some tests for the Pod modules hanging around in here that need to be moved out into *lib/*.

t/run/

Testing features of how perl actually runs, including exit codes and handling of PERL* environment variables.

The core uses the same testing style as the rest of Perl, a simple "ok/not ok" run through Test::Harness, but there are a few special considerations.

There are three ways to write a test in the core. Test::More, t/test.pl and ad hoc `print $test ? "ok 42\n" : "not ok 42\n"`. The decision of which to use depends on what part of the test suite you're working on. This is a measure to prevent a high-level failure (such as Config.pm breaking) from causing basic functionality tests to fail.

t/base t/comp

Since we don't know if `require` works, or even subroutines, use ad hoc tests for these two. Step carefully to avoid using the feature being tested.

t/cmd t/run t/io t/op

Now that basic `require()` and subroutines are tested, you can use the `t/test.pl` library which emulates the important features of Test::More while using a minimum of core features.

You can also conditionally use certain libraries like Config, but be sure to skip the test gracefully if it's not there.

t/lib ext lib

Now that the core of Perl is tested, Test::More can be used. You can also use the full suite of core modules in the tests.

When you say "make test" Perl uses the *t/TEST* program to run the test suite. All tests are run from the *t/* directory, **not** the directory which contains the test. This causes some problems with the tests in *lib/*, so here's some opportunity for some patching.

You must be triply conscious of cross-platform concerns. This usually boils down to using File::Spec and avoiding things like `fork()` and `system()` unless absolutely necessary.

Special Make Test Targets

There are various special make targets that can be used to test Perl slightly differently than the standard "test" target. Not all them are expected to give a 100% success rate. Many of them have several aliases.

coretest

Run *perl* on all core tests (*t/** and *lib/[a-z]** pragma tests).

test.depars

Run all the tests through the B::Deparse. Not all tests will succeed.

minitest

Run *miniperl* on *t/base*, *t/comp*, *t/cmd*, *t/run*, *t/io*, *t/op*, and *t/uni* tests.

test.third check.third utest.third ucheck.third

(Only in Tru64) Run all the tests using the memory leak + naughty memory access tool "Third Degree". The log files will be named *perl3.log.testname*.

test.torture torturetest

Run all the usual tests and some extra tests. As of Perl 5.8.0 the only extra tests are Abigail's JAPHs, *t/japh/abigail.t*.

You can also run the torture test with *t/harness* by giving `-torture` argument to *t/harness*.

utest ucheck test.utf8 check.utf8

Run all the tests with `-Mutf8`. Not all tests will succeed.

EXTERNAL TOOLS FOR DEBUGGING PERL

Sometimes it helps to use external tools while debugging and testing Perl. This section tries to guide you through using some common testing and debugging tools with Perl. This is meant as a guide to interfacing these tools with Perl, not as any kind of guide to the use of the tools themselves.

Rational Software's Purify

Purify is a commercial tool that is helpful in identifying memory overruns, wild pointers, memory leaks and other such badness. Perl must be compiled in a specific way for optimal testing with Purify. Purify is available under Windows NT, Solaris, HP-UX, SGI, and Siemens Unix.

The only currently known leaks happen when there are compile-time errors within eval or require. (Fixing these is non-trivial, unfortunately, but they must be fixed eventually.)

Purify on Unix

On Unix, Purify creates a new Perl binary. To get the most benefit out of Purify, you should create the perl to Purify using:

```
sh Configure -Accflags=-DPURIFY -Doptimize='-g' \
-Uusemymalloc -Dusemultiplicity
```

where these arguments mean:

-Accflags=-DPURIFY

Disables Perl's arena memory allocation functions, as well as forcing use of memory allocation functions derived from the system malloc.

-Doptimize='-g'

Adds debugging information so that you see the exact source statements where the problem occurs. Without this flag, all you will see is the source filename of where the error occurred.

-Uusemymalloc

Disable Perl's malloc so that Purify can more closely monitor allocations and leaks. Using Perl's malloc will make Purify report most leaks in the "potential" leaks category.

-Dusemultiplicity

Enabling the multiplicity option allows perl to clean up thoroughly when the interpreter shuts down, which reduces the number of bogus leak reports from Purify.

Once you've compiled a perl suitable for Purify'ing, then you can just:

```
make pureperl
```

which creates a binary named 'pureperl' that has been Purify'ed. This binary is used in place of the standard 'perl' binary when you want to debug Perl memory problems.

To minimize the number of memory leak false alarms (see [/PERL_DESTRUCT_LEVEL](#)), set environment variable PERL_DESTRUCT_LEVEL to 2.

```
setenv PERL_DESTRUCT_LEVEL 2
```

In Bourne-type shells:

```
PERL_DESTRUCT_LEVEL=2
export PERL_DESTRUCT_LEVEL
```

As an example, to show any memory leaks produced during the standard Perl testset you would create and run the Purify'ed perl as:

```
make pureperl
cd t
../pureperl -I../lib harness
```


which would run Perl on test.pl and report any memory problems.

Purify outputs messages in "Viewer" windows by default. If you don't have a windowing environment or if you simply want the Purify output to unobtrusively go to a log file instead of to the interactive window, use these following options to output to the log file "perl.log":

```
setenv PURIFYOPTIONS "-chain-length=25 -windows=no \  
-log-file=perl.log -append-logfile=yes"
```

If you plan to use the "Viewer" windows, then you only need this option:

```
setenv PURIFYOPTIONS "-chain-length=25"
```

In Bourne-type shells:

```
PURIFYOPTIONS="..."  
export PURIFYOPTIONS
```

or if you have the "env" utility:

```
env PURIFYOPTIONS="..." ../pureperl ...
```

Purify on NT

Purify on Windows NT instruments the Perl binary 'perl.exe' on the fly. There are several options in the makefile you should change to get the most use out of Purify:

DEFINES

You should add `-DPURIFY` to the `DEFINES` line so the `DEFINES` line looks something like:

```
DEFINES = -DWIN32 -D_CONSOLE -DNO_STRICT $(CRYPT_FLAG) -DPURIFY=1
```

to disable Perl's arena memory allocation functions, as well as to force use of memory allocation functions derived from the system malloc.

USE_MULTI = define

Enabling the multiplicity option allows perl to clean up thoroughly when the interpreter shuts down, which reduces the number of bogus leak reports from Purify.

#PERL_MALLOC = define

Disable Perl's malloc so that Purify can more closely monitor allocations and leaks. Using Perl's malloc will make Purify report most leaks in the "potential" leaks category.

CFG = Debug

Adds debugging information so that you see the exact source statements where the problem occurs. Without this flag, all you will see is the source filename of where the error occurred.

As an example, to show any memory leaks produced during the standard Perl testset you would create and run Purify as:

```
cd win32  
make  
cd ../t  
purify ../perl -I../lib harness
```

which would instrument Perl in memory, run Perl on test.pl, then finally report any memory problems.

NOTE: as of Perl 5.8.0, the `ext/Encode/t/Unicode.t` takes extraordinarily long (hours?) to complete under Purify. It has been theorized that it would eventually finish, but nobody has so far been patient enough :-). (This same extreme slowdown has been seen also with the Third Degree tool, so the said test must be doing something that is quite unfriendly for memory debuggers.) It is suggested that you simply kill away that testing process.

Compaq's/Digital's/HP's Third Degree

Third Degree is a tool for memory leak detection and memory access checks. It is one of the many tools in the ATOM toolkit. The toolkit is only available on Tru64 (formerly known as Digital UNIX formerly known as DEC OSF/1).

When building Perl, you must first run Configure with `-Doptimize=-g` and `-Uusemymalloc` flags, after that you can use the make targets `"perl.third"` and `"test.third"`. (What is required is that Perl must be compiled using the `-g` flag, you may need to re-Configure.)

The short story is that with `"atom"` you can instrument the Perl executable to create a new executable called *perl.third*. When the instrumented executable is run, it creates a log of dubious memory traffic in file called *perl.3log*. See the manual pages of `atom` and `third` for more information. The most extensive Third Degree documentation is available in the Compaq "Tru64 UNIX Programmer's Guide", chapter "Debugging Programs with Third Degree".

The `"test.third"` leaves a lot of files named *foo_bar.3log* in the `t/` subdirectory. There is a problem with these files: Third Degree is so effective that it finds problems also in the system libraries. Therefore you should use the `Porting/thirdclean` script to cleanup the **.3log* files.

There are also leaks that for given certain definition of a leak, aren't. See [/PERL_DESTRUCT_LEVEL](#) for more information.

PERL_DESTRUCT_LEVEL

If you want to run any of the tests yourself manually using the `pureperl` or `perl.third` executables, please note that by default perl **does not** explicitly cleanup all the memory it has allocated (such as global memory arenas) but instead lets the `exit()` of the whole program "take care" of such allocations, also known as "global destruction of objects".

There is a way to tell perl to do complete cleanup: set the environment variable `PERL_DESTRUCT_LEVEL` to a non-zero value. The `t/TEST` wrapper does set this to 2, and this is what you need to do too, if you don't want to see the "global leaks": For example, for "third-degreed" Perl:

```
env PERL_DESTRUCT_LEVEL=2 ./perl.third -Ilib t/foo/bar.t
```

(Note: the `mod_perl` apache module uses also this environment variable for its own purposes and extended its semantics. Refer to the `mod_perl` documentation for more information.)

Profiling

Depending on your platform there are various of profiling Perl.

There are two commonly used techniques of profiling executables: *statistical time-sampling* and *basic-block counting*.

The first method takes periodically samples of the CPU program counter, and since the program counter can be correlated with the code generated for functions, we get a statistical view of in which functions the program is spending its time. The caveats are that very small/fast functions have lower probability of showing up in the profile, and that periodically interrupting the program (this is usually done rather frequently, in the scale of milliseconds) imposes an additional overhead that may skew the results. The first problem can be alleviated by running the code for longer (in general this is a good idea for profiling), the second problem is usually kept in guard by the profiling tools themselves.

The second method divides up the generated code into *basic blocks*. Basic blocks are sections of code that are entered only in the beginning and exited only at the end. For example, a conditional jump starts a basic block. Basic block profiling usually works by *instrumenting* the code by adding *enter basic block #nnnn* book-keeping code to the generated code. During the execution of the code the basic block counters are then updated appropriately. The caveat is that the added extra code can skew the results: again, the profiling tools usually try to factor their own effects out of the results.

Gprof Profiling

gprof is a profiling tool available in many UNIX platforms, it uses *statistical time-sampling*.

You can build a profiled version of perl called "perl.gprof" by invoking the make target "perl.gprof" (What is required is that Perl must be compiled using the `-pg` flag, you may need to re-Configure). Running the profiled version of Perl will create an output file called **gmon.out** is created which contains the profiling data collected during the execution.

The gprof tool can then display the collected data in various ways. Usually gprof understands the following options:

- a Suppress statically defined functions from the profile.
- b Suppress the verbose descriptions in the profile.
- e routine
Exclude the given routine and its descendants from the profile.
- f routine
Display only the given routine and its descendants in the profile.
- s Generate a summary file called **gmon.sum** which then may be given to subsequent gprof runs to accumulate data over several runs.
- z Display routines that have zero usage.

For more detailed explanation of the available commands and output formats, see your own local documentation of gprof.

GCC gcov Profiling

Starting from GCC 3.0 *basic block profiling* is officially available for the GNU CC.

You can build a profiled version of perl called **perl.gcov** by invoking the make target "perl.gcov" (what is required that Perl must be compiled using gcc with the flags `-fprofile-arcs -ftest-coverage`, you may need to re-Configure).

Running the profiled version of Perl will cause profile output to be generated. For each source file an accompanying ".da" file will be created.

To display the results you use the "gcov" utility (which should be installed if you have gcc 3.0 or newer installed). **gcov** is run on source code files, like this

```
gcov sv.c
```

which will cause **sv.c.gcov** to be created. The **.gcov** files contain the source code annotated with relative frequencies of execution indicated by "#" markers.

Useful options of **gcov** include `-b` which will summarise the basic block, branch, and function call coverage, and `-c` which instead of relative frequencies will use the actual counts. For more information on the use of **gcov** and basic block profiling with gcc, see the latest GNU CC manual, as of GCC 3.0 see

<http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc.html>

and its section titled "8. gcov: a Test Coverage Program"

http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html#SEC132

Pixie Profiling

Pixie is a profiling tool available on IRIX and Tru64 (aka Digital UNIX aka DEC OSF/1) platforms. Pixie does its profiling using *basic-block counting*.

You can build a profiled version of perl called **perl.pixie** by invoking the make target "perl.pixie" (what is required is that Perl must be compiled using the `-g` flag, you may need to re-Configure).

In Tru64 a file called *perl.Addrs* will also be silently created, this file contains the addresses of the basic blocks. Running the profiled version of Perl will create a new file called "perl.Counts" which contains the counts for the basic block for that particular program execution.

To display the results you use the *prof* utility. The exact incantation depends on your operating system, "prof perl.Counts" in IRIX, and "prof -pixie -all -L. perl" in Tru64.

In IRIX the following prof options are available:

- h Reports the most heavily used lines in descending order of use. Useful for finding the hotspot lines.
- l Groups lines by procedure, with procedures sorted in descending order of use. Within a procedure, lines are listed in source order. Useful for finding the hotspots of procedures.

In Tru64 the following options are available:

- p[rocedures]
Procedures sorted in descending order by the number of cycles executed in each procedure. Useful for finding the hotspot procedures. (This is the default option.)
- h[eavy]
Lines sorted in descending order by the number of cycles executed in each line. Useful for finding the hotspot lines.
- i[nvocations]
The called procedures are sorted in descending order by number of calls made to the procedures. Useful for finding the most used procedures.
- l[ines]
Grouped by procedure, sorted by cycles executed per procedure. Useful for finding the hotspots of procedures.
- testcoverage
The compiler emitted code for these lines, but the code was unexecuted.
- z[ero]
Unexecuted procedures.

For further information, see your system's manual pages for pixie and prof.

Miscellaneous tricks

- Those debugging perl with the DDD frontend over gdb may find the following useful:
You can extend the data conversion shortcuts menu, so for example you can display an SV's IV value with one click, without doing any typing. To do that simply edit ~/.ddd/init file and add after:

```
! Display shortcuts.
Ddd*gdbDisplayShortcuts: \
/t () // Convert to Bin\n\
/d () // Convert to Dec\n\
/x () // Convert to Hex\n\
/o () // Convert to Oct(\n\
```

the following two lines:

```
((XPV*) (())->sv_any )->xpv_pv // 2pvx\n\
((XPVIV*) (())->sv_any )->xiv_iv // 2ivx
```

so now you can do ivx and pvx lookups or you can plug there the sv_peek "conversion":

```
Perl_sv_peek(my_perl, (SV*) ()) // sv_peek
```

(The `my_perl` is for threaded builds.) Just remember that every line, but the last one, should end with `\n`

Alternatively edit the init file interactively via: 3rd mouse button – New Display – Edit Menu

Note: you can define up to 20 conversion shortcuts in the `gdb` section.

- If you see in a debugger a memory area mysteriously full of `0xabababab`, you may be seeing the effect of the `Poison()` macro, see [perlclib](#).

CONCLUSION

We've had a brief look around the Perl source, an overview of the stages *perl* goes through when it's running your code, and how to use a debugger to poke at the Perl guts. We took a very simple problem and demonstrated how to solve it fully – with documentation, regression tests, and finally a patch for submission to p5p. Finally, we talked about how to use external tools to debug and test Perl.

I'd now suggest you read over those references again, and then, as soon as possible, get your hands dirty. The best way to learn is by doing, so:

- Subscribe to `perl5-porters`, follow the patches and try and understand them; don't be afraid to ask if there's a portion you're not clear on – who knows, you may unearth a bug in the patch...
- Keep up to date with the bleeding edge Perl distributions and get familiar with the changes. Try and get an idea of what areas people are working on and the changes they're making.
- Do read the README associated with your operating system, e.g. `README.aix` on the IBM AIX OS. Don't hesitate to supply patches to that README if you find anything missing or changed over a new OS release.
- Find an area of Perl that seems interesting to you, and see if you can work out how it works. Scan through the source, and step over it in the debugger. Play, poke, investigate, fiddle! You'll probably get to understand not just your chosen area but a much wider range of *perl*'s activity as well, and probably sooner than you'd think.

The Road goes ever on and on, down from the door where it began.

If you can do these things, you've started on the long road to Perl porting. Thanks for wanting to help make Perl better – and happy hacking!

AUTHOR

This document was written by Nathan Torkington, and is maintained by the `perl5-porters` mailing list.

NAME

```
perlhist – the Perl history records
=begin RCS
# # $Id: perlhist.pod,v 1.2 2000/01/24 11:44:47 jhi Exp $ #
=end RCS
```

DESCRIPTION

This document aims to record the Perl source code releases.

INTRODUCTION

Perl history in brief, by Larry Wall:

```
Perl 0 introduced Perl to my officemates.
Perl 1 introduced Perl to the world, and changed /\(...\|...\)/ to
/(\...|\...)/. \ (Dan Faigin still hasn't forgiven me. :-\ )
Perl 2 introduced Henry Spencer's regular expression package.
Perl 3 introduced the ability to handle binary data (embedded nulls).
Perl 4 introduced the first Camel book. Really. We mostly just
switched version numbers so the book could refer to 4.000.
Perl 5 introduced everything else, including the ability to
introduce everything else.
```

THE KEEPERS OF THE PUMPKIN

Larry Wall, Andy Dougherty, Tom Christiansen, Charles Bailey, Nick Ing-Simmons, Chip Salzenberg, Tim Bunce, Malcolm Beattie, Gurusamy Sarathy, Graham Barr, Jarkko Hietaniemi.

PUMPKIN?

[from Porting/pumpkin.pod in the Perl source code distribution]

Chip Salzenberg gets credit for that, with a nod to his cow orker, David Croy. We had passed around various names (baton, token, hot potato) but none caught on. Then, Chip asked:

[begin quote]

Who has the patch pumpkin?

To explain: David Croy once told me once that at a previous job, there was one tape drive and multiple systems that used it for backups. But instead of some high-tech exclusion software, they used a low-tech method to prevent multiple simultaneous backups: a stuffed pumpkin. No one was allowed to make backups unless they had the "backup pumpkin".

[end quote]

The name has stuck. The holder of the pumpkin is sometimes called the pumpking (keeping the source afloat?) or the pumpkineer (pulling the strings?).

THE RECORDS

Pump-	Release	Date	Notes
king			(by no means comprehensive, see Changes* for details)

```
=====
Larry  0          Classified.    Don't ask.
Larry  1.000      1987-Dec-18
```

	1.001..10	1988-Jan-30	
	1.011..14	1988-Feb-02	
Larry	2.000	1988-Jun-05	
	2.001	1988-Jun-28	
Larry	3.000	1989-Oct-18	
	3.001	1989-Oct-26	
	3.002..4	1989-Nov-11	
	3.005	1989-Nov-18	
	3.006..8	1989-Dec-22	
	3.009..13	1990-Mar-02	
	3.014	1990-Mar-13	
	3.015	1990-Mar-14	
	3.016..18	1990-Mar-28	
	3.019..27	1990-Aug-10	User subs.
	3.028	1990-Aug-14	
	3.029..36	1990-Oct-17	
	3.037	1990-Oct-20	
	3.040	1990-Nov-10	
	3.041	1990-Nov-13	
	3.042..43	1991-Jan-??	
	3.044	1991-Jan-12	
Larry	4.000	1991-Mar-21	
	4.001..3	1991-Apr-12	
	4.004..9	1991-Jun-07	
	4.010	1991-Jun-10	
	4.011..18	1991-Nov-05	
	4.019	1991-Nov-11	Stable.
	4.020..33	1992-Jun-08	
	4.034	1992-Jun-11	
	4.035	1992-Jun-23	
Larry	4.036	1993-Feb-05	Very stable.
	5.000alpha1	1993-Jul-31	
	5.000alpha2	1993-Aug-16	
	5.000alpha3	1993-Oct-10	
	5.000alpha4	1993-???-??	
	5.000alpha5	1993-???-??	
	5.000alpha6	1994-Mar-18	
	5.000alpha7	1994-Mar-25	
Andy	5.000alpha8	1994-Apr-04	
Larry	5.000alpha9	1994-May-05	ext appears.
	5.000alpha10	1994-Jun-11	
	5.000alpha11	1994-Jul-01	
Andy	5.000a11a	1994-Jul-07	To fit 14.
	5.000a11b	1994-Jul-14	
	5.000a11c	1994-Jul-19	
	5.000a11d	1994-Jul-22	
Larry	5.000alpha12	1994-Aug-04	
Andy	5.000a12a	1994-Aug-08	
	5.000a12b	1994-Aug-15	
	5.000a12c	1994-Aug-22	
	5.000a12d	1994-Aug-22	

	5.000a12e	1994-Aug-22	
	5.000a12f	1994-Aug-24	
	5.000a12g	1994-Aug-24	
	5.000a12h	1994-Aug-24	
Larry	5.000beta1	1994-Aug-30	
Andy	5.000b1a	1994-Sep-06	
Larry	5.000beta2	1994-Sep-14	Core slushified.
Andy	5.000b2a	1994-Sep-14	
	5.000b2b	1994-Sep-17	
	5.000b2c	1994-Sep-17	
Larry	5.000beta3	1994-Sep-??	
Andy	5.000b3a	1994-Sep-18	
	5.000b3b	1994-Sep-22	
	5.000b3c	1994-Sep-23	
	5.000b3d	1994-Sep-27	
	5.000b3e	1994-Sep-28	
	5.000b3f	1994-Sep-30	
	5.000b3g	1994-Oct-04	
Andy	5.000b3h	1994-Oct-07	
Larry?	5.000gamma	1994-Oct-13?	
Larry	5.000	1994-Oct-17	
Andy	5.000a	1994-Dec-19	
	5.000b	1995-Jan-18	
	5.000c	1995-Jan-18	
	5.000d	1995-Jan-18	
	5.000e	1995-Jan-18	
	5.000f	1995-Jan-18	
	5.000g	1995-Jan-18	
	5.000h	1995-Jan-18	
	5.000i	1995-Jan-26	
	5.000j	1995-Feb-07	
	5.000k	1995-Feb-11	
	5.000l	1995-Feb-21	
	5.000m	1995-Feb-28	
	5.000n	1995-Mar-07	
	5.000o	1995-Mar-13?	
Larry	5.001	1995-Mar-13	
Andy	5.001a	1995-Mar-15	
	5.001b	1995-Mar-31	
	5.001c	1995-Apr-07	
	5.001d	1995-Apr-14	
	5.001e	1995-Apr-18	Stable.
	5.001f	1995-May-31	
	5.001g	1995-May-25	
	5.001h	1995-May-25	
	5.001i	1995-May-30	
	5.001j	1995-Jun-05	
	5.001k	1995-Jun-06	
	5.001l	1995-Jun-06	Stable.
	5.001m	1995-Jul-02	Very stable.
	5.001n	1995-Oct-31	Very unstable.
	5.002beta1	1995-Nov-21	

	5.002b1a	1995-Dec-04	
	5.002b1b	1995-Dec-04	
	5.002b1c	1995-Dec-04	
	5.002b1d	1995-Dec-04	
	5.002b1e	1995-Dec-08	
	5.002b1f	1995-Dec-08	
Tom	5.002b1g	1995-Dec-21	Doc release.
Andy	5.002b1h	1996-Jan-05	
	5.002b2	1996-Jan-14	
Larry	5.002b3	1996-Feb-02	
Andy	5.002gamma	1996-Feb-11	
Larry	5.002delta	1996-Feb-27	
Larry	5.002	1996-Feb-29	Prototypes.
Charles	5.002_01	1996-Mar-25	
	5.003	1996-Jun-25	Security release.
	5.003_01	1996-Jul-31	
Nick	5.003_02	1996-Aug-10	
Andy	5.003_03	1996-Aug-28	
	5.003_04	1996-Sep-02	
	5.003_05	1996-Sep-12	
	5.003_06	1996-Oct-07	
	5.003_07	1996-Oct-10	
Chip	5.003_08	1996-Nov-19	
	5.003_09	1996-Nov-26	
	5.003_10	1996-Nov-29	
	5.003_11	1996-Dec-06	
	5.003_12	1996-Dec-19	
	5.003_13	1996-Dec-20	
	5.003_14	1996-Dec-23	
	5.003_15	1996-Dec-23	
	5.003_16	1996-Dec-24	
	5.003_17	1996-Dec-27	
	5.003_18	1996-Dec-31	
	5.003_19	1997-Jan-04	
	5.003_20	1997-Jan-07	
	5.003_21	1997-Jan-15	
	5.003_22	1997-Jan-16	
	5.003_23	1997-Jan-25	
	5.003_24	1997-Jan-29	
	5.003_25	1997-Feb-04	
	5.003_26	1997-Feb-10	
	5.003_27	1997-Feb-18	
	5.003_28	1997-Feb-21	
	5.003_90	1997-Feb-25	Ramping up to the 5.004 release.
	5.003_91	1997-Mar-01	
	5.003_92	1997-Mar-06	
	5.003_93	1997-Mar-10	
	5.003_94	1997-Mar-22	
	5.003_95	1997-Mar-25	
	5.003_96	1997-Apr-01	
	5.003_97	1997-Apr-03	Fairly widely used.
	5.003_97a	1997-Apr-05	

	5.003_97b	1997-Apr-08	
	5.003_97c	1997-Apr-10	
	5.003_97d	1997-Apr-13	
	5.003_97e	1997-Apr-15	
	5.003_97f	1997-Apr-17	
	5.003_97g	1997-Apr-18	
	5.003_97h	1997-Apr-24	
	5.003_97i	1997-Apr-25	
	5.003_97j	1997-Apr-28	
	5.003_98	1997-Apr-30	
	5.003_99	1997-May-01	
	5.003_99a	1997-May-09	
	p54rc1	1997-May-12	Release Candidates.
	p54rc2	1997-May-14	
Chip	5.004	1997-May-15	A major maintenance release.
Tim	5.004_01-t1	1997-???-??	The 5.004 maintenance track.
	5.004_01-t2	1997-Jun-11	aka perl5.004m1t2
	5.004_01	1997-Jun-13	
	5.004_01_01	1997-Jul-29	aka perl5.004m2t1
	5.004_01_02	1997-Aug-01	aka perl5.004m2t2
	5.004_01_03	1997-Aug-05	aka perl5.004m2t3
	5.004_02	1997-Aug-07	
	5.004_02_01	1997-Aug-12	aka perl5.004m3t1
	5.004_03-t2	1997-Aug-13	aka perl5.004m3t2
	5.004_03	1997-Sep-05	
	5.004_04-t1	1997-Sep-19	aka perl5.004m4t1
	5.004_04-t2	1997-Sep-23	aka perl5.004m4t2
	5.004_04-t3	1997-Oct-10	aka perl5.004m4t3
	5.004_04-t4	1997-Oct-14	aka perl5.004m4t4
	5.004_04	1997-Oct-15	
	5.004_04-m1	1998-Mar-04	(5.004m5t1) Maint. trials for 5.004_05.
	5.004_04-m2	1998-May-01	
	5.004_04-m3	1998-May-15	
	5.004_04-m4	1998-May-19	
	5.004_05-MT5	1998-Jul-21	
	5.004_05-MT6	1998-Oct-09	
	5.004_05-MT7	1998-Nov-22	
	5.004_05-MT8	1998-Dec-03	
Chip	5.004_05-MT9	1999-Apr-26	
	5.004_05	1999-Apr-29	
Malcolm	5.004_50	1997-Sep-09	The 5.005 development track.
	5.004_51	1997-Oct-02	
	5.004_52	1997-Oct-15	
	5.004_53	1997-Oct-16	
	5.004_54	1997-Nov-14	
	5.004_55	1997-Nov-25	
	5.004_56	1997-Dec-18	
	5.004_57	1998-Feb-03	
	5.004_58	1998-Feb-06	
	5.004_59	1998-Feb-13	
	5.004_60	1998-Feb-20	
	5.004_61	1998-Feb-27	
	5.004_62	1998-Mar-06	

	5.004_63	1998-Mar-17	
	5.004_64	1998-Apr-03	
	5.004_65	1998-May-15	
	5.004_66	1998-May-29	
Sarathy	5.004_67	1998-Jun-15	
	5.004_68	1998-Jun-23	
	5.004_69	1998-Jun-29	
	5.004_70	1998-Jul-06	
	5.004_71	1998-Jul-09	
	5.004_72	1998-Jul-12	
	5.004_73	1998-Jul-13	
	5.004_74	1998-Jul-14	5.005 beta candidate.
	5.004_75	1998-Jul-15	5.005 beta1.
	5.004_76	1998-Jul-21	5.005 beta2.
	5.005	1998-Jul-22	Oneperl.
Sarathy	5.005_01	1998-Jul-27	The 5.005 maintenance track.
	5.005_02-T1	1998-Aug-02	
	5.005_02-T2	1998-Aug-05	
	5.005_02	1998-Aug-08	
Graham	5.005_03-MT1	1998-Nov-30	
	5.005_03-MT2	1999-Jan-04	
	5.005_03-MT3	1999-Jan-17	
	5.005_03-MT4	1999-Jan-26	
	5.005_03-MT5	1999-Jan-28	
	5.005_03-MT6	1999-Mar-05	
	5.005_03	1999-Mar-28	
Sarathy	5.005_50	1998-Jul-26	The 5.6 development track.
	5.005_51	1998-Aug-10	
	5.005_52	1998-Sep-25	
	5.005_53	1998-Oct-31	
	5.005_54	1998-Nov-30	
	5.005_55	1999-Feb-16	
	5.005_56	1999-Mar-01	
	5.005_57	1999-May-25	
	5.005_58	1999-Jul-27	
	5.005_59	1999-Aug-02	
	5.005_60	1999-Aug-02	
	5.005_61	1999-Aug-20	
	5.005_62	1999-Oct-15	
	5.005_63	1999-Dec-09	
	5.5.640	2000-Feb-02	
	5.5.650	2000-Feb-08	beta1
	5.5.660	2000-Feb-22	beta2
	5.5.670	2000-Feb-29	beta3
	5.6.0-RC1	2000-Mar-09	Release candidate 1.
	5.6.0-RC2	2000-Mar-14	Release candidate 2.
	5.6.0-RC3	2000-Mar-21	Release candidate 3.
	5.6.0	2000-Mar-22	
Sarathy	5.6.1-TRIAL1	2000-Dec-18	The 5.6 maintenance track.
	5.6.1-TRIAL2	2001-Jan-31	
	5.6.1-TRIAL3	2001-Mar-19	
	5.6.1-foolish	2001-Apr-01	The "fools-gold" release.
	5.6.1	2001-Apr-08	

Jarkko	5.7.0	2000-Sep-02	The 5.7 track: Development.
	5.7.1	2001-Apr-09	
	5.7.2	2001-Jul-13	Virtual release candidate 0.
	5.7.3	2002-Mar-05	
	5.8.0-RC1	2002-Jun-01	
	5.8.0-RC2	2002-Jun-21	
	5.8.0-RC3	2002-Jul-13	
	5.8.0	2002-Jul-18	

SELECTED RELEASE SIZES

For example the notation "core: 212 29" in the release 1.000 means that it had in the core 212 kilobytes, in 29 files. The "core".."doc" are explained below.

release	core		lib		ext		t		doc	
=====										
1.000	212	29	-	-	-	-	38	51	62	3
1.014	219	29	-	-	-	-	39	52	68	4
2.000	309	31	2	3	-	-	55	57	92	4
2.001	312	31	2	3	-	-	55	57	94	4
3.000	508	36	24	11	-	-	79	73	156	5
3.044	645	37	61	20	-	-	90	74	190	6
4.000	635	37	59	20	-	-	91	75	198	4
4.019	680	37	85	29	-	-	98	76	199	4
4.036	709	37	89	30	-	-	98	76	208	5
5.000alpha2	785	50	114	32	-	-	112	86	209	5
5.000alpha3	801	50	117	33	-	-	121	87	209	5
5.000alpha9	1022	56	149	43	116	29	125	90	217	6
5.000a12h	978	49	140	49	205	46	152	97	228	9
5.000b3h	1035	53	232	70	216	38	162	94	218	21
5.000	1038	53	250	76	216	38	154	92	536	62
5.001m	1071	54	388	82	240	38	159	95	544	29
5.002	1121	54	661	101	287	43	155	94	847	35
5.003	1129	54	680	102	291	43	166	100	853	35
5.003_07	1231	60	748	106	396	53	213	137	976	39
5.004	1351	60	1230	136	408	51	355	161	1587	55
5.004_01	1356	60	1258	138	410	51	358	161	1587	55
5.004_04	1375	60	1294	139	413	51	394	162	1629	55
5.004_05	1463	60	1435	150	394	50	445	175	1855	59
5.004_51	1401	61	1260	140	413	53	358	162	1594	56
5.004_53	1422	62	1295	141	438	70	394	162	1637	56
5.004_56	1501	66	1301	140	447	74	408	165	1648	57
5.004_59	1555	72	1317	142	448	74	424	171	1678	58
5.004_62	1602	77	1327	144	629	92	428	173	1674	58
5.004_65	1626	77	1358	146	615	92	446	179	1698	60
5.004_68	1856	74	1382	152	619	92	463	187	1784	60
5.004_70	1863	75	1456	154	675	92	494	194	1809	60
5.004_73	1874	76	1467	152	762	102	506	196	1883	61
5.004_75	1877	76	1467	152	770	103	508	196	1896	62
5.005	1896	76	1469	152	795	103	509	197	1945	63
5.005_03	1936	77	1541	153	813	104	551	201	2176	72
5.005_50	1969	78	1842	301	795	103	514	198	1948	63
5.005_53	1999	79	1885	303	806	104	602	224	2002	67
5.005_56	2086	79	1970	307	866	113	672	238	2221	75
5.6.0	2930	80	2626	364	1096	129	868	281	2841	93
5.7.0	2977	80	2801	425	1250	132	975	307	3206	100

5.6.1	3049	80	3764	484	1924	159	1025	304	3593	119
5.7.1	3351	84	3442	455	1944	167	1334	357	3698	124
5.7.2	3491	87	4858	618	3290	298	1598	449	3910	139
5.7.3	3415	87	5367	630	14448	410	2205	640	4491	148

The "core"... "doc" mean the following files from the Perl source code distribution. The glob notation ** means recursively, (.) means regular files.

```
core    *. [hcy]
lib      lib/**/*.[ml]
ext      ext/**/*.{ [hcyt],xs,pm}
t        t/**/*.(.) (for 1-5.005_56) or **/*.[t] (for 5.6.0-5.7.3)
doc      {README*,INSTALL,*[_.]man{,?.},pod/**/*.[pod]}
```

Here are some statistics for the other subdirectories and one file in the Perl source distribution for somewhat more selected releases.

Legend: kB #														
	1.014		2.001		3.044		4.000		4.019		4.036			
atarist	-	-	-	-	-	-	-	-	-	-	113	31		
Configure	31	1	37	1	62	1	73	1	83	1	86	1		
eg	-	-	34	28	47	39	47	39	47	39	47	39		
emacs	-	-	-	-	-	-	67	4	67	4	67	4		
h2pl	-	-	-	-	12	12	12	12	12	12	12	12		
hints	-	-	-	-	-	-	-	-	5	42	11	56		
msdos	-	-	-	-	41	13	57	15	58	15	60	15		
os2	-	-	-	-	63	22	81	29	81	29	113	31		
usub	-	-	-	-	21	16	25	7	43	8	43	8		
x2p	103	17	104	17	137	17	147	18	152	19	154	19		
=====														
	5.000a2		5.000a12h		5.000b3h		5.000		5.001m		5.002		5.003	
atarist	113	31	113	31	-	-	-	-	-	-	-	-	-	-
bench	-	-	0	1	-	-	-	-	-	-	-	-	-	-
Bugs	2	5	26	1	-	-	-	-	-	-	-	-	-	-
dlperl	40	5	-	-	-	-	-	-	-	-	-	-	-	-
do	127	71	-	-	-	-	-	-	-	-	-	-	-	-
Configure	-	-	153	1	159	1	160	1	180	1	201	1	201	1
Doc	-	-	26	1	75	7	11	1	11	1	-	-	-	-
eg	79	58	53	44	51	43	54	44	54	44	54	44	54	44
emacs	67	4	104	6	104	6	104	1	104	6	108	1	108	1
h2pl	12	12	12	12	12	12	12	12	12	12	12	12	12	12
hints	11	56	12	46	18	48	18	48	44	56	73	59	77	60
msdos	60	15	60	15	-	-	-	-	-	-	-	-	-	-
os2	113	31	113	31	-	-	-	-	-	-	84	17	56	10
U	-	-	62	8	112	42	-	-	-	-	-	-	-	-
usub	43	8	-	-	-	-	-	-	-	-	-	-	-	-
utils	-	-	-	-	-	-	-	-	-	-	87	7	88	7
vms	-	-	80	7	123	9	184	15	304	20	500	24	475	26
x2p	171	22	171	21	162	20	162	20	279	20	280	20	280	20
=====														
	5.003_07		5.004		5.004_04		5.004_62		5.004_65		5.004_68			

beos	-	-	-	-	-	-	-	1	1	1	1	
Configure	217	1	225	1	225	1	240	1	248	1	256	1
cygwin32	-	-	23	5	23	5	23	5	24	5	24	5
djgpp	-	-	-	-	-	-	14	5	14	5	14	5
eg	54	44	81	62	81	62	81	62	81	62	81	62
emacs	143	1	194	1	204	1	212	2	212	2	212	2
h2pl	12	12	12	12	12	12	12	12	12	12	12	12
hints	90	62	129	69	132	71	144	72	151	74	155	74
os2	117	42	121	42	127	42	127	44	129	44	129	44
plan9	79	15	82	15	82	15	82	15	82	15	82	15
Porting	51	1	94	2	109	4	203	6	234	8	241	9
qnx	-	-	1	2	1	2	1	2	1	2	1	2
utils	97	7	112	8	118	8	124	8	156	9	159	9
vms	505	27	518	34	524	34	538	34	569	34	569	34
win32	-	-	285	33	378	36	470	39	493	39	575	41
x2p	280	19	281	19	281	19	281	19	282	19	281	19

=====

	5.004_70	5.004_73	5.004_75	5.005	5.005_03					
apollo	-	-	-	-	0	1				
beos	1	1	1	1	1	1				
Configure	256	1	256	1	264	1	270	1		
cygwin32	24	5	24	5	24	5	24	5		
djgpp	14	5	14	5	14	5	14	5	15	5
eg	86	65	86	65	86	65	86	65	86	65
emacs	262	2	262	2	262	2	262	2	274	2
h2pl	12	12	12	12	12	12	12	12	12	12
hints	157	74	157	74	159	74	160	74	179	77
mint	-	-	-	-	-	-	-	-	4	7
mpeix	-	-	-	-	5	3	5	3	5	3
os2	129	44	139	44	142	44	143	44	148	44
plan9	82	15	82	15	82	15	82	15	82	15
Porting	241	9	253	9	259	10	264	12	272	13
qnx	1	2	1	2	1	2	1	2	1	2
utils	160	9	160	9	160	9	160	9	164	9
vms	570	34	572	34	573	34	575	34	583	34
vos	-	-	-	-	-	-	-	-	156	10
win32	577	41	585	41	585	41	587	41	600	42
x2p	281	19	281	19	281	19	281	19	281	19

SELECTED PATCH SIZES

The "diff lines kb" means that for example the patch 5.003_08, to be applied on top of the 5.003_07 (or whatever was before the 5.003_08) added lines for 110 kilobytes, it removed lines for 19 kilobytes, and changed lines for 424 kilobytes. Just the lines themselves are counted, not their context. The "+ - !" become from the diff(1) context diff output format.

Pump-	Release	Date	diff	lines	kb
king			+	-	!
=====					
Chip	5.003_08	1996-Nov-19	110	19	424
	5.003_09	1996-Nov-26	38	9	248
	5.003_10	1996-Nov-29	29	2	27
	5.003_11	1996-Dec-06	73	12	165

	5.003_12	1996-Dec-19	275	6	436
	5.003_13	1996-Dec-20	95	1	56
	5.003_14	1996-Dec-23	23	7	333
	5.003_15	1996-Dec-23	0	0	1
	5.003_16	1996-Dec-24	12	3	50
	5.003_17	1996-Dec-27	19	1	14
	5.003_18	1996-Dec-31	21	1	32
	5.003_19	1997-Jan-04	80	3	85
	5.003_20	1997-Jan-07	18	1	146
	5.003_21	1997-Jan-15	38	10	221
	5.003_22	1997-Jan-16	4	0	18
	5.003_23	1997-Jan-25	71	15	119
	5.003_24	1997-Jan-29	426	1	20
	5.003_25	1997-Feb-04	21	8	169
	5.003_26	1997-Feb-10	16	1	15
	5.003_27	1997-Feb-18	32	10	38
	5.003_28	1997-Feb-21	58	4	66
	5.003_90	1997-Feb-25	22	2	34
	5.003_91	1997-Mar-01	37	1	39
	5.003_92	1997-Mar-06	16	3	69
	5.003_93	1997-Mar-10	12	3	15
	5.003_94	1997-Mar-22	407	7	200
	5.003_95	1997-Mar-25	41	1	37
	5.003_96	1997-Apr-01	283	5	261
	5.003_97	1997-Apr-03	13	2	34
	5.003_97a	1997-Apr-05	57	1	27
	5.003_97b	1997-Apr-08	14	1	20
	5.003_97c	1997-Apr-10	20	1	16
	5.003_97d	1997-Apr-13	8	0	16
	5.003_97e	1997-Apr-15	15	4	46
	5.003_97f	1997-Apr-17	7	1	33
	5.003_97g	1997-Apr-18	6	1	42
	5.003_97h	1997-Apr-24	23	3	68
	5.003_97i	1997-Apr-25	23	1	31
	5.003_97j	1997-Apr-28	36	1	49
	5.003_98	1997-Apr-30	171	12	539
	5.003_99	1997-May-01	6	0	7
	5.003_99a	1997-May-09	36	2	61
	p54rc1	1997-May-12	8	1	11
	p54rc2	1997-May-14	6	0	40
	5.004	1997-May-15	4	0	4
Tim	5.004_01	1997-Jun-13	222	14	57
	5.004_02	1997-Aug-07	112	16	119
	5.004_03	1997-Sep-05	109	0	17
	5.004_04	1997-Oct-15	66	8	173

THE KEEPERS OF THE RECORDS

Jarkko Hietaniemi <jhi@iki.fi>.

Thanks to the collective memory of the Perlfolk. In addition to the Keepers of the Pumpkin also Alan Champion, Mark Dominus, Andreas König, John Macdonald, Matthias Neeracher, Jeff Okamoto, Michael Pepler, Randal Schwartz, and Paul D. Smith sent corrections and additions.

NAME

perlintern – autogenerated documentation of purely **internal**
Perl functions

DESCRIPTION

This file is the autogenerated documentation of functions in the Perl interpreter that are documented using Perl's internal documentation format but are not marked as part of the Perl API. In other words, **they are not for use in extensions!**

Global Variables**PL_DBsingle**

When Perl is run in debugging mode, with the `-d` switch, this SV is a boolean which indicates whether subs are being single-stepped. Single-stepping is automatically turned on after every step. This is the C variable which corresponds to Perl's `$DB::single` variable. See `PL_DBsub`.

SV * PL_DBsingle

=for hackers Found in file `intrpvar.h`

PL_DBsub

When Perl is run in debugging mode, with the `-d` switch, this GV contains the SV which holds the name of the sub being debugged. This is the C variable which corresponds to Perl's `$DB::sub` variable. See `PL_DBsingle`.

GV * PL_DBsub

=for hackers Found in file `intrpvar.h`

PL_DBtrace

Trace variable used when Perl is run in debugging mode, with the `-d` switch. This is the C variable which corresponds to Perl's `$DB::trace` variable. See `PL_DBsingle`.

SV * PL_DBtrace

=for hackers Found in file `intrpvar.h`

PL_dowarn

The C variable which corresponds to Perl's `$^W` warning variable.

bool PL_dowarn

=for hackers Found in file `intrpvar.h`

PL_last_in_gv

The GV which was last used for a filehandle input operation. (`<` `<FH`)

GV* PL_last_in_gv

=for hackers Found in file `thrdvar.h`

PL_ofs_sv

The output field separator – `$,` in Perl space.

SV* PL_ofs_sv

=for hackers Found in file `thrdvar.h`

PL_rs

The input record separator – `$/` in Perl space.

SV* PL_rs

=for hackers Found in file thrdvar.h

GV Functions

is_gv_magical

Returns TRUE if given the name of a magical GV.

Currently only useful internally when determining if a GV should be created even in rvalue contexts.

flags is not used at present but available for future extension to allow selecting particular classes of magical variable.

```
bool    is_gv_magical(char *name, STRLEN len, U32 flags)
```

=for hackers Found in file gv.c

IO Functions

start_glob

Function called by `do_readline` to spawn a glob (or do the glob inside perl on VMS). This code used to be inline, but now perl uses `File::Glob` this glob starter is only used by miniperl during the build process. Moving it away shrinks `pp_hot.c`; shrinking `pp_hot.c` helps speed perl up.

```
PerlIO* start_glob(SV* pattern, IO *io)
```

=for hackers Found in file doio.c

Pad Data Structures

CvPADLIST

CV's can have `CvPADLIST(cv)` set to point to an AV.

For these purposes "forms" are a kind-of CV, `eval""`s are too (except they're not callable at will and are always thrown away after the `eval""` is done executing).

XSUBs don't have `CvPADLIST` set – `dxSTARG` fetches values from `PL_curpad`, but that is really the callers pad (a slot of which is allocated by every `entersub`).

The `CvPADLIST` AV has does not have `AvREAL` set, so `REFCNT` of component items is managed "manual" (mostly in `op.c`) rather than normal `av.c` rules. The items in the AV are not SVs as for a normal AV, but other AVs:

0'th Entry of the `CvPADLIST` is an AV which represents the "names" or rather the "static type information" for lexicals.

The `CvDEPTH`'th entry of `CvPADLIST` AV is an AV which is the stack frame at that depth of recursion into the CV. The 0'th slot of a frame AV is an AV which is `@_`. other entries are storage for variables and op targets.

During compilation: `PL_comppad_name` is set the the the names AV. `PL_comppad` is set the the frame AV for the frame `CvDEPTH == 1`. `PL_curpad` is set the body of the frame AV (i.e. `AvARRAY(PL_comppad)`).

Iterating over the names AV iterates over all possible pad items. Pad slots that are `SVs_PADTMP` (targets/GVs/constants) end up having `&PL_sv_undef` "names" (see `pad_alloc()`).

Only my/our variable (`SVs_PADMY`/`SVs_PADOOR`) slots get valid names. The rest are op targets/GVs/constants which are statically allocated or resolved at compile time. These don't have names by which they can be looked up from Perl code at run time through `eval""` like my/our variables can be. Since they can't be looked up by "name" but only by their index allocated at compile time (which is usually in `PL_op-op_targ`), wasting a name SV for them

doesn't make sense.

The SVs in the names AV have their PV being the name of the variable. NV+1..IV inclusive is a range of cop_seq numbers for which the name is valid. For typed lexicals name SV is SVt_PVMG and SvSTASH points at the type.

If SvFAKE is set on the name SV then slot in the frame AVs are a REFCNT'ed references to a lexical from "outside".

If the 'name' is '&' the the corresponding entry in frame AV is a CV representing a possible closure. (SvFAKE and name of '&' is not a meaningful combination currently but could become so if my sub foo { } is implemented.)

```
AV *      CvPADLIST(CV *cv)
```

=for hackers Found in file cv.h

Stack Manipulation Macros

djSP Declare Just SP. This is actually identical to dSP, and declares a local copy of perl's stack pointer, available via the SP macro. See SP. (Available for backward source code compatibility with the old (Perl 5.005) thread model.)

```
      djSP;
```

=for hackers Found in file pp.h

LVRET True if this op will be the return value of an lvalue subroutine

=for hackers Found in file pp.h

SV Manipulation Functions

report_uninit

Print appropriate "Use of uninitialized variable" warning

```
      void      report_uninit()
```

=for hackers Found in file sv.c

sv_add_arena

Given a chunk of memory, link it to the head of the list of arenas, and split it into a list of free SVs.

```
      void      sv_add_arena(char* ptr, U32 size, U32 flags)
```

=for hackers Found in file sv.c

sv_clean_all

Decrement the refcnt of each remaining SV, possibly triggering a cleanup. This function may have to be called multiple times to free SVs which are in complex self-referential hierarchies.

```
      I32      sv_clean_all()
```

=for hackers Found in file sv.c

sv_clean_objs

Attempt to destroy all objects not yet freed

```
      void      sv_clean_objs()
```

=for hackers Found in file sv.c

sv_free_arenas

Deallocate the memory used by all arenas. Note that all the individual SV heads and bodies within the arenas must already have been freed.

```
void    sv_free_arenas()
```

=for hackers Found in file sv.c

AUTHORS

The autodocumentation system was originally added to the Perl core by Benjamin Stuhl. Documentation is by whoever was kind enough to document their functions.

SEE ALSO

perlgets(1), perlapi(1)

NAME

perlintro — a brief introduction and overview of Perl

DESCRIPTION

This document is intended to give you a quick overview of the Perl programming language, along with pointers to further documentation. It is intended as a "bootstrap" guide for those who are new to the language, and provides just enough information for you to be able to read other peoples' Perl and understand roughly what it's doing, or write your own simple scripts.

This introductory document does not aim to be complete. It does not even aim to be entirely accurate. In some cases perfection has been sacrificed in the goal of getting the general idea across. You are *strongly* advised to follow this introduction with more information from the full Perl manual, the table of contents to which can be found in [perltoc](#).

Throughout this document you'll see references to other parts of the Perl documentation. You can read that documentation using the `perldoc` command or whatever method you're using to read this document.

What is Perl?

Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more.

The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features are that it's easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing, and has one of the world's most impressive collections of third-party modules.

Different definitions of Perl are given in [perl](#), [perlfaq1](#) and no doubt other places. From this we can determine that Perl is different things to different people, but that lots of people think it's at least worth writing about.

Running Perl programs

To run a Perl program from the Unix command line:

```
perl progname.pl
```

Alternatively, put this as the first line of your script:

```
#!/usr/bin/env perl
```

... and run the script as `/path/to/script.pl`. Of course, it'll need to be executable first, so `chmod 755 script.pl` (under Unix).

For more information, including instructions for other platforms such as Windows and Mac OS, read [perlrun](#).

Basic syntax overview

A Perl script or program consists of one or more statements. These statements are simply written in the script in a straightforward fashion. There is no need to have a `main()` function or anything of that kind.

Perl statements end in a semi-colon:

```
print "Hello, world";
```

Comments start with a hash symbol and run to the end of the line

```
# This is a comment
```

Whitespace is irrelevant:

```
print
    "Hello, world"
;
```

... except inside quoted strings:

```
# this would print with a linebreak in the middle
print "Hello
world";
```

Double quotes or single quotes may be used around literal strings:

```
print "Hello, world";
print 'Hello, world';
```

However, only double quotes "interpolate" variables and special characters such as newlines (`\n`):

```
print "Hello, $name\n";      # works fine
print 'Hello, $name\n';     # prints $name\n literally
```

Numbers don't need quotes around them:

```
print 42;
```

You can use parentheses for functions' arguments or omit them according to your personal taste. They are only required occasionally to clarify issues of precedence.

```
print("Hello, world\n");
print "Hello, world\n";
```

More detailed information about Perl syntax can be found in [perlsyn](#).

Perl variable types

Perl has three main variable types: scalars, arrays, and hashes.

Scalars

A scalar represents a single value:

```
my $animal = "camel";
my $answer = 42;
```

Scalar values can be strings, integers or floating point numbers, and Perl will automatically convert between them as required. There is no need to pre-declare your variable types.

Scalar values can be used in various ways:

```
print $animal;
print "The animal is $animal\n";
print "The square of $answer is ", $answer * $answer, "\n";
```

There are a number of "magic" scalars with names that look like punctuation or line noise. These special variables are used for all kinds of purposes, and are documented in [perlvar](#). The only one you need to know about for now is `$_` which is the "default variable". It's used as the default argument to a number of functions in Perl, and it's set implicitly by certain looping constructs.

```
print;          # prints contents of $_ by default
```

Arrays

An array represents a list of values:

```
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);
```

Arrays are zero-indexed. Here's how you get at elements in an array:

```
print $animals[0];          # prints "camel"
print $animals[1];          # prints "llama"
```

The special variable `$#array` tells you the index of the last element of an array:

```
print $mixed[$#mixed];           # last element, prints 1.23
```

You might be tempted to use `$#array + 1` to tell you how many items there are in an array. Don't bother. As it happens, using `@array` where Perl expects to find a scalar value ("in scalar context") will give you the number of elements in the array:

```
if (@animals < 5) { ... }
```

The elements we're getting from the array start with a `$` because we're getting just a single value out of the array — you ask for a scalar, you get a scalar.

To get multiple values from an array:

```
@animals[0,1];                  # gives ("camel", "llama");
@animals[0..2];                  # gives ("camel", "llama", "owl");
@animals[1..$#animals];          # gives all except the first element
```

This is called an "array slice".

You can do various useful things to lists:

```
my @sorted    = sort @animals;
my @backwards = reverse @numbers;
```

There are a couple of special arrays too, such as `@ARGV` (the command line arguments to your script) and `@_` (the arguments passed to a subroutine). These are documented in [perlvar](#).

Hashes

A hash represents a set of key/value pairs:

```
my %fruit_color = ("apple", "red", "banana", "yellow");
```

You can use whitespace and the `< =` operator to lay them out more nicely:

```
my %fruit_color = (
    apple => "red",
    banana => "yellow",
);
```

To get at hash elements:

```
$fruit_color{"apple"};           # gives "red"
```

You can get at lists of keys and values with `keys()` and `values()`.

```
my @fruits = keys %fruit_colors;
my @colors = values %fruit_colors;
```

Hashes have no particular internal order, though you can sort the keys and loop through them.

Just like special scalars and arrays, there are also special hashes. The most well known of these is `%ENV` which contains environment variables. Read all about it (and other special variables) in [perlvar](#).

Scalars, arrays and hashes are documented more fully in [perldata](#).

More complex data types can be constructed using references, which allow you to build lists and hashes within lists and hashes.

A reference is a scalar value and can refer to any other Perl data type. So by storing a reference as the value of an array or hash element, you can easily create lists and hashes within lists and hashes. The following example shows a 2 level hash of hash structure using anonymous hash references.

```
my $variables = {
```

```

    scalar => {
        description => "single item",
        sigil => '$',
    },
    array  => {
        description => "ordered list of items",
        sigil => '@',
    },
    hash   => {
        description => "key/value pairs",
        sigil => '%',
    },
};

print "Scalars begin with a $variables->{'scalar'}->{'sigil'}\n";

```

Exhaustive information on the topic of references can be found in [perlreftut](#), [perlrlol](#), [perlref](#) and [perldsc](#).

Variable scoping

Throughout the previous section all the examples have used the syntax:

```
my $var = "value";
```

The `my` is actually not required; you could just use:

```
$var = "value";
```

However, the above usage will create global variables throughout your program, which is bad programming practice. `my` creates lexically scoped variables instead. The variables are scoped to the block (i.e. a bunch of statements surrounded by curly-braces) in which they are defined.

```

my $a = "foo";
if ($some_condition) {
    my $b = "bar";
    print $a;           # prints "foo"
    print $b;           # prints "bar"
}
print $a;               # prints "foo"
print $b;               # prints nothing; $b has fallen out of scope

```

Using `my` in combination with a `use strict;` at the top of your Perl scripts means that the interpreter will pick up certain common programming errors. For instance, in the example above, the final `print $b` would cause a compile-time error and prevent you from running the program. Using `strict` is highly recommended.

Conditional and looping constructs

Perl has most of the usual conditional and looping constructs except for `case/switch` (but if you really want it, there is a `Switch` module in Perl 5.8 and newer, and on CPAN. See the section on modules, below, for more information about modules and CPAN).

The conditions can be any Perl expression. See the list of operators in the next section for information on comparison and boolean logic operators, which are commonly used in conditional statements.

```

if
    if ( condition ) {
        ...
    } elsif ( other condition ) {
        ...
    } else {
        ...
    }

```

```
}
```

There's also a negated version of it:

```
unless ( condition ) {
    ...
}
```

This is provided as a more readable version of `if (!condition)`.

Note that the braces are required in Perl, even if you've only got one line in the block. However, there is a clever way of making your one-line conditional blocks more English like:

```
# the traditional way
if ($zippy) {
    print "Yow!";
}

# the Perlsh post-condition way
print "Yow!" if $zippy;
print "We have no bananas" unless $bananas;
```

while

```
while ( condition ) {
    ...
}
```

There's also a negated version, for the same reason we have `unless`:

```
until ( condition ) {
    ...
}
```

You can also use `while` in a post-condition:

```
print "LA LA LA\n" while 1;           # loops forever
```

for Exactly like C:

```
for ($i=0; $i <= $max; $i++) {
    ...
}
```

The C style for loop is rarely needed in Perl since Perl provides the more friendly list scanning `foreach` loop.

foreach

```
foreach (@array) {
    print "This element is $_\n";
}

# you don't have to use the default $_ either...
foreach my $key (keys %hash) {
    print "The value of $key is $hash{$key}\n";
}
```

For more detail on looping constructs (and some that weren't mentioned in this overview) see [perlsyn](#).

Builtin operators and functions

Perl comes with a wide selection of builtin functions. Some of the ones we've already seen include `print`, `sort` and `reverse`. A list of them is given at the start of [perlfunc](#) and you can easily read about any given function by using `perldoc -f functionname`.

Perl operators are documented in full in [perlop](#), but here are a few of the most common ones:

Arithmetic

+	addition
-	subtraction
*	multiplication
/	division

Numeric comparison

==	equality
!=	inequality
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal

String comparison

eq	equality
ne	inequality
lt	less than
gt	greater than
le	less than or equal
ge	greater than or equal

(Why do we have separate numeric and string comparisons? Because we don't have special variable types, and Perl needs to know whether to sort numerically (where 99 is less than 100) or alphabetically (where 100 comes before 99).)

Boolean logic

&&	and
	or
!	not

(and, or and not aren't just in the above table as descriptions of the operators — they're also supported as operators in their own right. They're more readable than the C-style operators, but have different precedence to && and friends. Check [perlop](#) for more detail.)

Miscellaneous

=	assignment
.	string concatenation
x	string multiplication
..	range operator (creates a list of numbers)

Many operators can be combined with a = as follows:

\$a += 1;	# same as \$a = \$a + 1
\$a -= 1;	# same as \$a = \$a - 1
\$a .= "\n";	# same as \$a = \$a . "\n";

Files and I/O

You can open a file for input or output using the `open()` function. It's documented in extravagant detail in [perlfunc](#) and [perlopentut](#), but in short:

```
open(INFILE, "input.txt") or die "Can't open input.txt: $!";
open(OUTFILE, ">output.txt") or die "Can't open output.txt: $!";
open(LOGFILE, ">>my.log") or die "Can't open logfile: $!";
```

You can read from an open filehandle using the `<<` operator. In scalar context it reads a single line from the filehandle, and in list context it reads the whole file in, assigning each line to an element of the list:

```
my $line = <INFILE>;
my @lines = <INFILE>;
```

Reading in the whole file at one time is called slurping. It can be useful but it may be a memory hog. Most text file processing can be done a line at a time with Perl's looping constructs.

The `<` operator is most often seen in a while loop:

```
while (<INFILE>) {      # assigns each line in turn to $_
    print "Just read in this line: $_";
}
```

We've already seen how to print to standard output using `print()`. However, `print()` can also take an optional first argument specifying which filehandle to print to:

```
print STDERR "This is your final warning.\n";
print OUTFILE $record;
print LOGFILE $logmessage;
```

When you're done with your filehandles, you should `close()` them (though to be honest, Perl will clean up after you if you forget):

```
close INFILE;
```

Regular expressions

Perl's regular expression support is both broad and deep, and is the subject of lengthy documentation in [perlrequick](#), [perlretut](#), and elsewhere. However, in short:

Simple matching

```
if (/foo/)      { ... } # true if $_ contains "foo"
if ($a =~ /foo/) { ... } # true if $a contains "foo"
```

The `//` matching operator is documented in [perlop](#). It operates on `$_` by default, or can be bound to another variable using the `=~` binding operator (also documented in [perlop](#)).

Simple substitution

```
s/foo/bar/;          # replaces foo with bar in $_
$a =~ s/foo/bar/;    # replaces foo with bar in $a
$a =~ s/foo/bar/g;    # replaces ALL INSTANCES of foo with bar in $a
```

The `s///` substitution operator is documented in [perlop](#).

More complex regular expressions

You don't just have to match on fixed strings. In fact, you can match on just about anything you could dream of by using more complex regular expressions. These are documented at great length in [perlre](#), but for the meantime, here's a quick cheat sheet:

<code>.</code>	a single character
<code>\s</code>	a whitespace character (space, tab, newline)
<code>\S</code>	non-whitespace character
<code>\d</code>	a digit (0-9)
<code>\D</code>	a non-digit
<code>\w</code>	a word character (a-z, A-Z, 0-9, _)
<code>\W</code>	a non-word character
<code>[aeiou]</code>	matches a single character in the given set
<code>[^aeiou]</code>	matches a single character outside the given set
<code>(foo bar baz)</code>	matches any of the alternatives specified
<code>^</code>	start of string
<code>\$</code>	end of string

Quantifiers can be used to specify how many of the previous thing you want to match on, where "thing" means either a literal character, one of the metacharacters listed above, or a group of characters or metacharacters in parentheses.

<code>*</code>	zero or more of the previous thing
<code>+</code>	one or more of the previous thing
<code>?</code>	zero or one of the previous thing
<code>{3}</code>	matches exactly 3 of the previous thing
<code>{3,6}</code>	matches between 3 and 6 of the previous thing
<code>{3,}</code>	matches 3 or more of the previous thing

Some brief examples:

<code>/^\d+/</code>	string starts with one or more digits
<code>/^\$/</code>	nothing in the string (start and end are adjacent)
<code>/(\d\s){3}/</code>	a three digits, each followed by a whitespace character (eg "3 4 5 ")
<code>/(a.+)/</code>	matches a string in which every odd-numbered letter is a (eg "abacadaf")

```
# This loop reads from STDIN, and prints non-blank lines:
while (<>) {
    next if /^$/;
    print;
}
```

Parentheses for capturing

As well as grouping, parentheses serve a second purpose. They can be used to capture the results of parts of the regexp match for later use. The results end up in `$1`, `$2` and so on.

```
# a cheap and nasty way to break an email address up into parts
if ($email =~ /^([^\s]*)+@(\.+)/) {
    print "Username is $1\n";
    print "Hostname is $2\n";
}
```

Other regexp features

Perl regexps also support backreferences, lookaheads, and all kinds of other complex details. Read all about them in [perlrequick](#), [perlretut](#), and [perlre](#).

Writing subroutines

Writing subroutines is easy:

```
sub log {
    my $logmessage = shift;
    print LOGFILE $logmessage;
}
```

What's that `shift`? Well, the arguments to a subroutine are available to us as a special array called `@_` (see [perlvar](#) for more on that). The default argument to the `shift` function just happens to be `@_`. So `my $logmessage = shift;` shifts the first item off the list of arguments and assigns it to `$logmessage`.

We can manipulate `@_` in other ways too:

```
my ($logmessage, $priority) = @_;      # common
my $logmessage = $_[0];                # uncommon, and ugly
```

Subroutines can also return values:

```
sub square {  
    my $num = shift;  
    my $result = $num * $num;  
    return $result;  
}
```

For more information on writing subroutines, see [perlsub](#).

OO Perl

OO Perl is relatively simple and is implemented using references which know what sort of object they are based on Perl's concept of packages. However, OO Perl is largely beyond the scope of this document. Read [perlboot](#), [perltoot](#), [perltooc](#) and [perlobj](#).

As a beginning Perl programmer, your most common use of OO Perl will be in using third-party modules, which are documented below.

Using Perl modules

Perl modules provide a range of features to help you avoid reinventing the wheel, and can be downloaded from CPAN (<http://www.cpan.org/>). A number of popular modules are included with the Perl distribution itself.

Categories of modules range from text manipulation to network protocols to database integration to graphics. A categorized list of modules is also available from CPAN.

To learn how to install modules you download from CPAN, read [perlmodinstall](#)

To learn how to use a particular module, use `perldoc Module:::Name`. Typically you will want to use `Module:::Name`, which will then give you access to exported functions or an OO interface to the module.

[perlfaq](#) contains questions and answers related to many common tasks, and often provides suggestions for good CPAN modules to use.

[perlmod](#) describes Perl modules in general. [perlmodlib](#) lists the modules which came with your Perl installation.

If you feel the urge to write Perl modules, [perlnewmod](#) will give you good advice.

AUTHOR

Kirrily "Skud" Robert <skud@cpan.org>

NAME

perliol – C API for Perl's implementation of IO in Layers.

SYNOPSIS

```
/* Defining a layer ... */
#include <perliol.h>
```

DESCRIPTION

This document describes the behavior and implementation of the PerlIO abstraction described in [perlapi](#) when `USE_PERLIO` is defined (and `USE_SFIO` is not).

History and Background

The PerlIO abstraction was introduced in perl5.003_02 but languished as just an abstraction until perl5.7.0. However during that time a number of perl extensions switched to using it, so the API is mostly fixed to maintain (source) compatibility.

The aim of the implementation is to provide the PerlIO API in a flexible and platform neutral manner. It is also a trial of an "Object Oriented C, with vtables" approach which may be applied to perl6.

Layers vs Disciplines

Initial discussion of the ability to modify IO streams behaviour used the term "discipline" for the entities which were added. This came (I believe) from the use of the term in "sfio", which in turn borrowed it from "line disciplines" on Unix terminals. However, this document (and the C code) uses the term "layer".

This is, I hope, a natural term given the implementation, and should avoid connotations that are inherent in earlier uses of "discipline" for things which are rather different.

Data Structures

The basic data structure is a PerlIOI:

```
typedef struct _PerlIO PerlIOI;
typedef struct _PerlIO_funcs PerlIO_funcs;
typedef PerlIOI *PerlIO;

struct _PerlIO
{
    PerlIOI *      next;          /* Lower layer */
    PerlIO_funcs * tab;          /* Functions for this layer */
    IV             flags;        /* Various flags for state */
};
```

A `PerlIOI *` is a pointer to the struct, and the *application* level `PerlIO *` is a pointer to a `PerlIOI *` – i.e. a pointer to a pointer to the struct. This allows the application level `PerlIO *` to remain constant while the actual `PerlIOI *` underneath changes. (Compare perl's `SV *` which remains constant while its `sv_any` field changes as the scalar's type changes.) An IO stream is then in general represented as a pointer to this linked-list of "layers".

It should be noted that because of the double indirection in a `PerlIO *`, a `< &(perlio-next)` "is" a `PerlIO *`, and so to some degree at least one layer can use the "standard" API on the next layer down.

A "layer" is composed of two parts:

1. The functions and attributes of the "layer class".
2. The per-instance data for a particular handle.

Functions and Attributes

The functions and attributes are accessed via the "tab" (for table) member of `PerlIOI`. The functions (methods of the layer "class") are fixed, and are defined by the `PerlIO_funcs` type. They are broadly the same as the public `PerlIO_XXXXX` functions:

```

struct _PerlIO_funcs
{
    Size_t      fsize;
    char *      name;
    Size_t      size;
    IV          kind;
    IV          (*Pushed)(pTHX_ PerlIO *f, const char *mode, SV *arg, PerlIO_funcs *tab);
    IV          (*Popped)(pTHX_ PerlIO *f);
    PerlIO *(*Open)(pTHX_ PerlIO_funcs *tab,
                    AV *layers, IV n,
                    const char *mode,
                    int fd, int imode, int perm,
                    PerlIO *old,
                    int narg, SV **args);
    IV          (*Binmode)(pTHX_ PerlIO *f);
    SV *        (*Getarg)(pTHX_ PerlIO *f, CLONE_PARAMS *param, int flags);
    IV          (*Fileno)(pTHX_ PerlIO *f);
    PerlIO *    (*Dup)(pTHX_ PerlIO *f, PerlIO *o, CLONE_PARAMS *param, int flags);
    /* Unix-like functions - cf sfio line disciplines */
    SSize_t(*Read)(pTHX_ PerlIO *f, void *vbuf, Size_t count);
    SSize_t(*Unread)(pTHX_ PerlIO *f, const void *vbuf, Size_t count);
    SSize_t(*Write)(pTHX_ PerlIO *f, const void *vbuf, Size_t count);
    IV          (*Seek)(pTHX_ PerlIO *f, Off_t offset, int whence);
    Off_t(*Tell)(pTHX_ PerlIO *f);
    IV          (*Close)(pTHX_ PerlIO *f);
    /* Stdio-like buffered IO functions */
    IV          (*Flush)(pTHX_ PerlIO *f);
    IV          (*Fill)(pTHX_ PerlIO *f);
    IV          (*Eof)(pTHX_ PerlIO *f);
    IV          (*Error)(pTHX_ PerlIO *f);
    void        (*Clearerr)(pTHX_ PerlIO *f);
    void        (*Setlinebuf)(pTHX_ PerlIO *f);
    /* Perl's snooping functions */
    STDCHAR *   (*Get_base)(pTHX_ PerlIO *f);
    Size_t(*Get_bufsiz)(pTHX_ PerlIO *f);
    STDCHAR *   (*Get_ptr)(pTHX_ PerlIO *f);
    SSize_t(*Get_cnt)(pTHX_ PerlIO *f);
    void        (*Set_ptrcnt)(pTHX_ PerlIO *f, STDCHAR *ptr, SSize_t cnt);
};

```

The first few members of the struct give a function table size for compatibility check "name" for the layer, the size to malloc for the per-instance data, and some flags which are attributes of the class as whole (such as whether it is a buffering layer), then follow the functions which fall into four basic groups:

1. Opening and setup functions
2. Basic IO operations
3. Stdio class buffering options.
4. Functions to support Perl's traditional "fast" access to the buffer.

A layer does not have to implement all the functions, but the whole table has to be present. Unimplemented slots can be NULL (which will result in an error when called) or can be filled in with stubs to "inherit" behaviour from a "base class". This "inheritance" is fixed for all instances of the layer, but as the layer chooses which stubs to populate the table, limited "multiple inheritance" is possible.

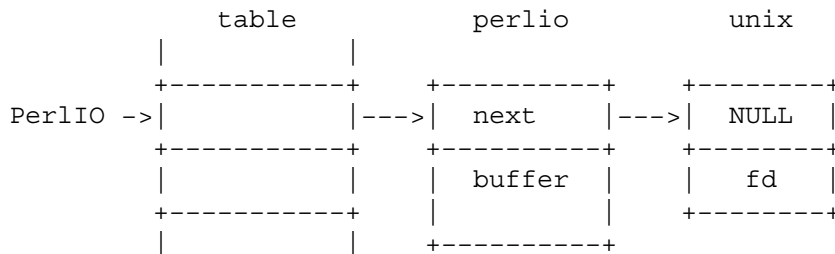
Per-instance Data

The per-instance data are held in memory beyond the basic PerlIOl struct, by making a PerlIOl the first member of the layer's struct thus:

```
typedef struct
{
    struct _PerlIO base;      /* Base "class" info */
    STDCHAR *    buf;        /* Start of buffer */
    STDCHAR *    end;        /* End of valid part of buffer */
    STDCHAR *    ptr;        /* Current position in buffer */
    Off_t        posn;        /* Offset of buf into the file */
    Size_t       bufsiz;     /* Real size of buffer */
    IV           oneword;     /* Emergency buffer */
} PerlIOBuf;
```

In this way (as for perl's scalars) a pointer to a PerlIOBuf can be treated as a pointer to a PerlIOl.

Layers in action.



The above attempts to show how the layer scheme works in a simple case. The application's `PerlIO *` points to an entry in the table(s) representing open (allocated) handles. For example the first three slots in the table correspond to `stdin`, `stdout` and `stderr`. The table in turn points to the current "top" layer for the handle – in this case an instance of the generic buffering layer "perlio". That layer in turn points to the next layer down – in this case the lowlevel "unix" layer.

The above is roughly equivalent to a "stdio" buffered stream, but with much more flexibility:

- If Unix level `read/write/lseek` is not appropriate for (say) sockets then the "unix" layer can be replaced (at open time or even dynamically) with a "socket" layer.
- Different handles can have different buffering schemes. The "top" layer could be the "mmap" layer if reading disk files was quicker using `mmap` than `read`. An "unbuffered" stream can be implemented simply by not having a buffer layer.
- Extra layers can be inserted to process the data as it flows through. This was the driving need for including the scheme in perl 5.7.0+ – we needed a mechanism to allow data to be translated between perl's internal encoding (conceptually at least Unicode as UTF-8), and the "native" format used by the system. This is provided by the `":encoding(xxxx)"` layer which typically sits above the buffering layer.
- A layer can be added that does `"\n"` to CRLF translation. This layer can be used on any platform, not just those that normally do such things.

Per-instance flag bits

The generic flag bits are a hybrid of `O_XXXXX` style flags deduced from the mode string passed to `PerlIO_open()`, and state bits for typical buffer layers.

`PERLIO_F_EOF`

End of file.

PERLIO_F_CANWRITE

Writes are permitted, i.e. opened as "w" or "r+" or "a", etc.

PERLIO_F_CANREAD

Reads are permitted i.e. opened "r" or "w+" (or even "a+" – ick).

PERLIO_F_ERROR

An error has occurred (for `PerlIO_error()`).

PERLIO_F_TRUNCATE

Truncate file suggested by open mode.

PERLIO_F_APPEND

All writes should be appends.

PERLIO_F_CRLF

Layer is performing Win32-like "\n" mapped to CR,LF for output and CR,LF mapped to "\n" for input. Normally the provided "crlf" layer is the only layer that need bother about this.

`PerlIO_binmode()` will mess with this flag rather than add/remove layers if the `PERLIO_K_CANCRLF` bit is set for the layers class.

PERLIO_F_UTF8

Data written to this layer should be UTF-8 encoded; data provided by this layer should be considered UTF-8 encoded. Can be set on any layer by ":utf8" dummy layer. Also set on ":encoding" layer.

PERLIO_F_UNBUF

Layer is unbuffered – i.e. write to next layer down should occur for each write to this layer.

PERLIO_F_WRBUF

The buffer for this layer currently holds data written to it but not sent to next layer.

PERLIO_F_RDBUF

The buffer for this layer currently holds unconsumed data read from layer below.

PERLIO_F_LINEBUF

Layer is line buffered. Write data should be passed to next layer down whenever a "\n" is seen. Any data beyond the "\n" should then be processed.

PERLIO_F_TEMP

File has been `unlink()`ed, or should be deleted on `close()`.

PERLIO_F_OPEN

Handle is open.

PERLIO_F_FASTGETS

This instance of this layer supports the "fast gets" interface. Normally set based on `PERLIO_K_FASTGETS` for the class and by the existence of the function(s) in the table. However a class that normally provides that interface may need to avoid it on a particular instance. The "pending" layer needs to do this when it is pushed above a layer which does not support the interface. (Perl's `sv_gets()` does not expect the streams fast gets behaviour to change during one "get".)

Methods in Detail**fsize**

```
Size_t fsize;
```

Size of the function table. This is compared against the value PerlIO code "knows" as a compatibility check. Future versions *may* be able to tolerate layers compiled against an old version of the headers.

name

```
char * name;
```

The name of the layer whose `open()` method Perl should invoke on `open()`. For example if the layer is called APR, you will call:

```
open $fh, ">:APR", ...
```

and Perl knows that it has to invoke the `PerlIOAPR_open()` method implemented by the APR layer.

size

```
Size_t size;
```

The size of the per-instance data structure, e.g.:

```
sizeof(PerlIOAPR)
```

If this field is zero then `PerlIO_pushed` does not malloc anything and assumes layer's `Pushed` function will do any required layer stack manipulation – used to avoid malloc/free overhead for dummy layers. If the field is non-zero it must be at least the size of `PerlIOl`, `PerlIO_pushed` will allocate memory for the layer's data structures and link new layer onto the stream's stack. (If the layer's `Pushed` method returns an error indication the layer is popped again.)

kind

```
IV kind;
```

- **PERLIO_K_BUFFERED**

The layer is buffered.

- **PERLIO_K_RAW**

The layer is acceptable to have in a `binmode(FH)` stack – i.e. it does not (or will configure itself not to) transform bytes passing through it.

- **PERLIO_K_CANCRLF**

Layer can translate between `"\n"` and CRLF line ends.

- **PERLIO_K_FASTGETS**

Layer allows buffer snooping.

- **PERLIO_K_MULTIARG**

Used when the layer's `open()` accepts more arguments than usual. The extra arguments should come not before the `MODE` argument. When this flag is used it's up to the layer to validate the args.

Pushed

```
IV (*Pushed)(pTHX_ PerlIO *f, const char *mode, SV *arg);
```

The only absolutely mandatory method. Called when the layer is pushed onto the stack. The `mode` argument may be `NULL` if this occurs post-open. The `arg` will be non-`NULL` if an argument string was passed. In most cases this should call `PerlIOBase_pushed()` to convert `mode` into the appropriate `PERLIO_F_XXXXX` flags in addition to any actions the layer itself takes. If a layer is not expecting an argument it need neither save the one passed to it, nor provide `Getarg()` (it could perhaps `Perl_warn` that the argument was un-expected).

Returns 0 on success. On failure returns -1 and should set `errno`.

Popped

```
IV (*Popped)(pTHX_ PerlIO *f);
```

Called when the layer is popped from the stack. A layer will normally be popped after `Close()` is

called. But a layer can be popped without being closed if the program is dynamically managing layers on the stream. In such cases `Popped()` should free any resources (buffers, translation tables, ...) not held directly in the layer's struct. It should also `Unread()` any unconsumed data that has been read and buffered from the layer below back to that layer, so that it can be re-provided to what ever is now above.

Returns 0 on success and failure.

Open

```
PerlIO *      (*Open) (...);
```

The `Open()` method has lots of arguments because it combines the functions of perl's `open`, `PerlIO_open`, perl's `sysopen`, `PerlIO_fdopen` and `PerlIO_reopen`. The full prototype is as follows:

```
PerlIO *      (*Open) (pTHX_ PerlIO_funcs *tab,
                      AV *layers, IV n,
                      const char *mode,
                      int fd, int imode, int perm,
                      PerlIO *old,
                      int nargs, SV **args);
```

`Open` should (perhaps indirectly) call `PerlIO_allocate()` to allocate a slot in the table and associate it with the layers information for the opened file, by calling `PerlIO_push`. The *layers* AV is an array of all the layers destined for the `PerlIO *`, and any arguments passed to them, *n* is the index into that array of the layer being called. The macro `PerlIOArg` will return a (possibly NULL) `SV *` for the argument passed to the layer.

The *mode* string is an "fopen()-like" string which would match the regular expression `/^[I#]?[rwa]\+?[bt]?$/`.

The 'I' prefix is used during creation of `stdin..stderr` via special `PerlIO_fdopen` calls; the '#' prefix means that this is `sysopen` and that *imode* and *perm* should be passed to `PerlLIO_open3`; 'r' means **read**, 'w' means **write** and 'a' means **append**. The '+' suffix means that both reading and writing/appending are permitted. The 'b' suffix means file should be binary, and 't' means it is text. (Almost all layers should do the IO in binary mode, and ignore the b/t bits. The `:crlf` layer should be pushed to handle the distinction.)

If *old* is not NULL then this is a `PerlIO_reopen`. Perl itself does not use this (yet?) and semantics are a little vague.

If *fd* not negative then it is the numeric file descriptor *fd*, which will be open in a manner compatible with the supplied mode string, the call is thus equivalent to `PerlLIO_fdopen`. In this case *nargs* will be zero.

If *nargs* is greater than zero then it gives the number of arguments passed to `open`, otherwise it will be 1 if for example `PerlLIO_open` was called. In simple cases `SvPV_nolen(*args)` is the pathname to `open`.

Having said all that translation-only layers do not need to provide `Open()` at all, but rather leave the opening to a lower level layer and wait to be "pushed". If a layer does provide `Open()` it should normally call the `Open()` method of next layer down (if any) and then push itself on top if that succeeds.

Returns NULL on failure.

Binmode

```
IV          (*Binmode) (pTHX_ PerlIO *f);
```

Optional. Used when `:raw` layer is pushed (explicitly or as a result of `binmode(FH)`). If not present layer will be popped. If present should configure layer as binary (or pop itself) and return 0. If it

returns `-1` for error `binmode` will fail with layer still on the stack.

Getarg

```
SV *      (*Getarg) (pTHX_ PerlIO *f,
                    CLONE_PARAMS *param, int flags);
```

Optional. If present should return an `SV *` representing the string argument passed to the layer when it was pushed. e.g. `":encoding(ascii)"` would return an `SvPV` with value `"ascii"`. (*param* and *flags* arguments can be ignored in most cases)

Fileno

```
IV      (*Fileno) (pTHX_ PerlIO *f);
```

Returns the Unix/Posix numeric file descriptor for the handle. Normally `PerlIOBase_fileno()` (which just asks next layer down) will suffice for this.

Returns `-1` on error, which is considered to include the case where the layer cannot provide such a file descriptor.

Dup

```
PerlIO * (*Dup) (pTHX_ PerlIO *f, PerlIO *o,
                 CLONE_PARAMS *param, int flags);
```

XXX: Needs more docs.

Used as part of the "clone" process when a thread is spawned (in which case *param* will be non-NULL) and when a stream is being duplicated via `'&'` in the open.

Similar to `Open`, returns `PerlIO*` on success, `NULL` on failure.

Read

```
SSize_t (*Read) (pTHX_ PerlIO *f, void *vbuf, Size_t count);
```

Basic read operation.

Typically will call `Fill` and manipulate pointers (possibly via the API). `PerlIOBuf_read()` may be suitable for derived classes which provide "fast gets" methods.

Returns actual bytes read, or `-1` on an error.

Unread

```
SSize_t (*Unread) (pTHX_ PerlIO *f,
                   const void *vbuf, Size_t count);
```

A superset of `stdio's ungetc()`. Should arrange for future reads to see the bytes in *vbuf*. If there is no obviously better implementation then `PerlIOBase_unread()` provides the function by pushing a "fake" "pending" layer above the calling layer.

Returns the number of unread chars.

Write

```
SSize_t (*Write) (PerlIO *f, const void *vbuf, Size_t count);
```

Basic write operation.

Returns bytes written or `-1` on an error.

Seek

```
IV      (*Seek) (pTHX_ PerlIO *f, Off_t offset, int whence);
```

Position the file pointer. Should normally call its own `Flush` method and then the `Seek` method of next layer down.

Returns `0` on success, `-1` on failure.

Tell

```
Off_t      (*Tell) (pTHX_ PerlIO *f);
```

Return the file pointer. May be based on layers cached concept of position to avoid overhead.

Returns -1 on failure to get the file pointer.

Close

```
IV         (*Close) (pTHX_ PerlIO *f);
```

Close the stream. Should normally call `PerlIOBase_close()` to flush itself and close layers below, and then deallocate any data structures (buffers, translation tables, ...) not held directly in the data structure.

Returns 0 on success, -1 on failure.

Flush

```
IV         (*Flush) (pTHX_ PerlIO *f);
```

Should make stream's state consistent with layers below. That is, any buffered write data should be written, and file position of lower layers adjusted for data read from below but not actually consumed. (Should perhaps `Unread()` such data to the lower layer.)

Returns 0 on success, -1 on failure.

Fill

```
IV         (*Fill) (pTHX_ PerlIO *f);
```

The buffer for this layer should be filled (for read) from layer below. When you "subclass" `PerlIOBuf` layer, you want to use its `_read` method and to supply your own fill method, which fills the `PerlIOBuf`'s buffer.

Returns 0 on success, -1 on failure.

Eof

```
IV         (*Eof) (pTHX_ PerlIO *f);
```

Return end-of-file indicator. `PerlIOBase_eof()` is normally sufficient.

Returns 0 on end-of-file, 1 if not end-of-file, -1 on error.

Error

```
IV         (*Error) (pTHX_ PerlIO *f);
```

Return error indicator. `PerlIOBase_error()` is normally sufficient.

Returns 1 if there is an error (usually when `PERLIO_F_ERROR` is set, 0 otherwise).

Clearerr

```
void       (*Clearerr) (pTHX_ PerlIO *f);
```

Clear end-of-file and error indicators. Should call `PerlIOBase_clearerr()` to set the `PERLIO_F_XXXXX` flags, which may suffice.

Setlinebuf

```
void       (*Setlinebuf) (pTHX_ PerlIO *f);
```

Mark the stream as line buffered. `PerlIOBase_setlinebuf()` sets the `PERLIO_F_LINEBUF` flag and is normally sufficient.

Get_base

```
STDCHAR *  (*Get_base) (pTHX_ PerlIO *f);
```

Allocate (if not already done so) the read buffer for this layer and return pointer to it. Return NULL on failure.

Get_bufsiz

```
Size_t (*Get_bufsiz)(pTHX_ PerlIO *f);
```

Return the number of bytes that last `Fill()` put in the buffer.

Get_ptr

```
STDCHAR * (*Get_ptr)(pTHX_ PerlIO *f);
```

Return the current read pointer relative to this layer's buffer.

Get_cnt

```
SSize_t (*Get_cnt)(pTHX_ PerlIO *f);
```

Return the number of bytes left to be read in the current buffer.

Set_ptrcnt

```
void (*Set_ptrcnt)(pTHX_ PerlIO *f,
                   STDCHAR *ptr, SSize_t cnt);
```

Adjust the read pointer and count of bytes to match `ptr` and/or `cnt`. The application (or layer above) must ensure they are consistent. (Checking is allowed by the paranoid.)

Core Layers

The file `perlio.c` provides the following layers:

"unix"

A basic non-buffered layer which calls Unix/POSIX `read()`, `write()`, `lseek()`, `close()`. No buffering. Even on platforms that distinguish between `O_TEXT` and `O_BINARY` this layer is always `O_BINARY`.

"perlio"

A very complete generic buffering layer which provides the whole of PerlIO API. It is also intended to be used as a "base class" for other layers. (For example its `Read()` method is implemented in terms of the `Get_cnt()/Get_ptr()/Set_ptrcnt()` methods).

"perlio" over "unix" provides a complete replacement for `stdio` as seen via PerlIO API. This is the default for `USE_PERLIO` when system's `stdio` does not permit perl's "fast gets" access, and which do not distinguish between `O_TEXT` and `O_BINARY`.

"stdio"

A layer which provides the PerlIO API via the layer scheme, but implements it by calling system's `stdio`. This is (currently) the default if system's `stdio` provides sufficient access to allow perl's "fast gets" access and which do not distinguish between `O_TEXT` and `O_BINARY`.

"crlf"

A layer derived using "perlio" as a base class. It provides Win32-like "\n" to CR,LF translation. Can either be applied above "perlio" or serve as the buffer layer itself. "crlf" over "unix" is the default if system distinguishes between `O_TEXT` and `O_BINARY` opens. (At some point "unix" will be replaced by a "native" Win32 IO layer on that platform, as Win32's read/write layer has various drawbacks.) The "crlf" layer is a reasonable model for a layer which transforms data in some way.

"mmap"

If `Configure` detects `mmap()` functions this layer is provided (with "perlio" as a "base") which does "read" operations by `mmap()`ing the file. Performance improvement is marginal on modern systems, so it is mainly there as a proof of concept. It is likely to be unbundled from the core at some point. The "mmap" layer is a reasonable model for a minimalist "derived" layer.

"pending"

An "internal" derivative of "perlio" which can be used to provide `Unread()` function for layers which have no buffer or cannot be bothered. (Basically this layer's `Fill()` pops itself off the stack and so resumes reading from layer below.)

"raw"

A dummy layer which never exists on the layer stack. Instead when "pushed" it actually pops the stack removing itself, it then calls `Binmode` function table entry on all the layers in the stack – normally this (via `PerlIOBase_binmode`) removes any layers which do not have `PERLIO_K_RAW` bit set. Layers can modify that behaviour by defining their own `Binmode` entry.

"utf8"

Another dummy layer. When pushed it pops itself and sets the `PERLIO_F_UTF8` flag on the layer which was (and now is once more) the top of the stack.

In addition *perlio.c* also provides a number of `PerlIOBase_xxxx()` functions which are intended to be used in the table slots of classes which do not need to do anything special for a particular method.

Extension Layers

Layers can be made available by extension modules. When an unknown layer is encountered the PerlIO code will perform the equivalent of:

```
use PerlIO 'layer';
```

Where *layer* is the unknown layer. *PerlIO.pm* will then attempt to:

```
require PerlIO::layer;
```

If after that process the layer is still not defined then the `open` will fail.

The following extension layers are bundled with perl:

":encoding"

```
use Encoding;
```

makes this layer available, although *PerlIO.pm* "knows" where to find it. It is an example of a layer which takes an argument as it is called thus:

```
open( $fh, "<:encoding(iso-8859-7)", $pathname );
```

":scalar"

Provides support for reading data from and writing data to a scalar.

```
open( $fh, "+<:scalar", \$scalar );
```

When a handle is so opened, then reads get bytes from the string value of *\$scalar*, and writes change the value. In both cases the position in *\$scalar* starts as zero but can be altered via `seek`, and determined via `tell`.

Please note that this layer is implied when calling `open()` thus:

```
open( $fh, "+<", \$scalar );
```

":via"

Provided to allow layers to be implemented as Perl code. For instance:

```
use PerlIO::via::StripHTML;
open( my $fh, "<:via(StripHTML)", "index.html" );
```

See *PerlIO::via* for details.

TODO

Things that need to be done to improve this document.

- Explain how to make a valid fh without going through `open()` (i.e. apply a layer). For example if the file is not opened through perl, but we want to get back a fh, like it was opened by Perl.

How `PerlIO_apply_layera` fits in, where its docs, was it made public?

Currently the example could be something like this:

```
PerlIO *foo_to_PerlIO(pTHX_ char *mode, ...)
{
    char *mode; /* "w", "r", etc */
    const char *layers = ":APR"; /* the layer name */
    PerlIO *f = PerlIO_allocate(aTHX);
    if (!f) {
        return NULL;
    }

    PerlIO_apply_layers(aTHX_ f, mode, layers);

    if (f) {
        PerlIOAPR *st = PerlIOSelf(f, PerlIOAPR);
        /* fill in the st struct, as in _open() */
        st->file = file;
        PerlIOBase(f)->flags |= PERLIO_F_OPEN;

        return f;
    }
    return NULL;
}
```

- fix/add the documentation in places marked as XXX.
- The handling of errors by the layer is not specified. e.g. when `$!` should be set explicitly, when the error handling should be just delegated to the top layer.

Probably give some hints on using `SETERRNO()` or pointers to where they can be found.

- I think it would help to give some concrete examples to make it easier to understand the API. Of course I agree that the API has to be concise, but since there is no second document that is more of a guide, I think that it'd make it easier to start with the doc which is an API, but has examples in it in places where things are unclear, to a person who is not a PerlIO guru (yet).

NAME

perlipc – Perl interprocess communication (signals, fifos, pipes, safe subprocesses, sockets, and semaphores)

DESCRIPTION

The basic IPC facilities of Perl are built out of the good old Unix signals, named pipes, pipe opens, the Berkeley socket routines, and SysV IPC calls. Each is used in slightly different situations.

Signals

Perl uses a simple signal handling model: the %SIG hash contains names or references of user-installed signal handlers. These handlers will be called with an argument which is the name of the signal that triggered it. A signal may be generated intentionally from a particular keyboard sequence like control-C or control-Z, sent to you from another process, or triggered automatically by the kernel when special events transpire, like a child process exiting, your process running out of stack space, or hitting file size limit.

For example, to trap an interrupt signal, set up a handler like this:

```
sub catch_zap {
    my $signame = shift;
    $shucks++;
    die "Somebody sent me a SIG$signame";
}
$SIG{INT} = 'catch_zap'; # could fail in modules
$SIG{INT} = \&catch_zap; # best strategy
```

Prior to Perl 5.7.3 it was necessary to do as little as you possibly could in your handler; notice how all we do is set a global variable and then raise an exception. That's because on most systems, libraries are not re-entrant; particularly, memory allocation and I/O routines are not. That meant that doing nearly *anything* in your handler could in theory trigger a memory fault and subsequent core dump – see [Deferred Signals](#) below.

The names of the signals are the ones listed out by `kill -l` on your system, or you can retrieve them from the Config module. Set up an @signame list indexed by number to get the name and a %signo table indexed by name to get the number:

```
use Config;
defined $Config{sig_name} || die "No sigs?";
foreach $name (split(' ', $Config{sig_name})) {
    $signo{$name} = $i;
    $signame[$i] = $name;
    $i++;
}
```

So to check whether signal 17 and SIGALRM were the same, do just this:

```
print "signal #17 = $signame[17]\n";
if ($signo{ALRM}) {
    print "SIGALRM is $signo{ALRM}\n";
}
```

You may also choose to assign the strings 'IGNORE' or 'DEFAULT' as the handler, in which case Perl will try to discard the signal or do the default thing.

On most Unix platforms, the CHLD (sometimes also known as CLD) signal has special behavior with respect to a value of 'IGNORE'. Setting \$SIG{CHLD} to 'IGNORE' on such a platform has the effect of not creating zombie processes when the parent process fails to wait() on its child processes (i.e. child processes are automatically reaped). Calling wait() with \$SIG{CHLD} set to 'IGNORE' usually returns -1 on such platforms.

Some signals can be neither trapped nor ignored, such as the KILL and STOP (but not the TSTP) signals. One strategy for temporarily ignoring signals is to use a `local()` statement, which will be automatically restored once your block is exited. (Remember that `local()` values are "inherited" by functions called from within that block.)

```
sub precious {
    local $SIG{INT} = 'IGNORE';
    &more_functions;
}
sub more_functions {
    # interrupts still ignored, for now...
}
```

Sending a signal to a negative process ID means that you send the signal to the entire Unix process-group. This code sends a hang-up signal to all processes in the current process group (and sets `$SIG{HUP}` to IGNORE so it doesn't kill itself):

```
{
    local $SIG{HUP} = 'IGNORE';
    kill HUP => -$$;
    # snazzy writing of: kill('HUP', -$$)
}
```

Another interesting signal to send is signal number zero. This doesn't actually affect another process, but instead checks whether it's alive or has changed its UID.

```
unless (kill 0 => $kid_pid) {
    warn "something wicked happened to $kid_pid";
}
```

You might also want to employ anonymous functions for simple signal handlers:

```
$SIG{INT} = sub { die "\nOutta here!\n" };
```

But that will be problematic for the more complicated handlers that need to reinstall themselves. Because Perl's signal mechanism is currently based on the `signal(3)` function from the C library, you may sometimes be so unfortunate as to run on systems where that function is "broken", that is, it behaves in the old unreliable SysV way rather than the newer, more reasonable BSD and POSIX fashion. So you'll see defensive people writing signal handlers like this:

```
sub REAPER {
    $waitedpid = wait;
    # loathe sysV: it makes us not only reinstate
    # the handler, but place it after the wait
    $SIG{CHLD} = \&REAPER;
}
$SIG{CHLD} = \&REAPER;
# now do something that forks...
```

or better still:

```
use POSIX ":sys_wait_h";
sub REAPER {
    my $child;
    # If a second child dies while in the signal handler caused by the
    # first death, we won't get another signal. So must loop here else
    # we will leave the unreaped child as a zombie. And the next time
    # two children die we get another zombie. And so on.
    while (($child = waitpid(-1, WNOHANG)) > 0) {
        $Kid_Status{$child} = $?;
```

```

    }
    $SIG{CHLD} = \&REAPER; # still loathe sysV
}
$SIG{CHLD} = \&REAPER;
# do something that forks...

```

Signal handling is also used for timeouts in Unix. While safely protected within an `eval{}` block, you set a signal handler to trap alarm signals and then schedule to have one delivered to you in some number of seconds. Then try your blocking operation, clearing the alarm when it's done but not before you've exited your `eval{}` block. If it goes off, you'll use `die()` to jump out of the block, much as you might using `longjmp()` or `throw()` in other languages.

Here's an example:

```

eval {
    local $SIG{ALRM} = sub { die "alarm clock restart" };
    alarm 10;
    flock(FH, 2); # blocking write lock
    alarm 0;
};
if ($@ and $@ !~ /alarm clock restart/) { die }

```

If the operation being timed out is `system()` or `qx()`, this technique is liable to generate zombies. If this matters to you, you'll need to do your own `fork()` and `exec()`, and kill the errant child process.

For more complex signal handling, you might see the standard POSIX module. Lamentably, this is almost entirely undocumented, but the *t/lib/posix.t* file from the Perl source distribution has some examples in it.

Handling the SIGHUP Signal in Daemons

A process that usually starts when the system boots and shuts down when the system is shut down is called a daemon (Disk And Execution MONitor). If a daemon process has a configuration file which is modified after the process has been started, there should be a way to tell that process to re-read its configuration file, without stopping the process. Many daemons provide this mechanism using the SIGHUP signal handler. When you want to tell the daemon to re-read the file you simply send it the SIGHUP signal.

Not all platforms automatically reinstall their (native) signal handlers after a signal delivery. This means that the handler works only the first time the signal is sent. The solution to this problem is to use POSIX signal handlers if available, their behaviour is well-defined.

The following example implements a simple daemon, which restarts itself every time the SIGHUP signal is received. The actual code is located in the subroutine `code()`, which simply prints some debug info to show that it works and should be replaced with the real code.

```

#!/usr/bin/perl -w

use POSIX ();
use FindBin ();
use File::Basename ();
use File::Spec::Functions;

$|=1;

# make the daemon cross-platform, so exec always calls the script
# itself with the right path, no matter how the script was invoked.
my $script = File::Basename::basename($0);
my $SELF = catfile $FindBin::Bin, $script;

# POSIX unmask the sigprocmask properly
my $sigset = POSIX::SigSet->new();
my $action = POSIX::SigAction->new('sighUP_handler',
                                     $sigset,

```

```

                                &POSIX::SA_NODEFER);
    POSIX::sigaction(&POSIX::SIGHUP, $action);

    sub sigHUP_handler {
        print "got SIGHUP\n";
        exec($SELF, @ARGV) or die "Couldn't restart: $!\n";
    }

    code();

    sub code {
        print "PID: $$\n";
        print "ARGV: @ARGV\n";
        my $c = 0;
        while (++$c) {
            sleep 2;
            print "$c\n";
        }
    }

    __END__

```

Named Pipes

A named pipe (often referred to as a FIFO) is an old Unix IPC mechanism for processes communicating on the same machine. It works just like a regular, connected anonymous pipes, except that the processes rendezvous using a filename and don't have to be related.

To create a named pipe, use the Unix command `mknod(1)` or on some systems, `mkfifo(1)`. These may not be in your normal path.

```

# system return val is backwards, so && not ||
#
$ENV{PATH} .= ":/etc:/usr/etc";
if (    system('mknod', $path, 'p')
      && system('mkfifo', $path) )
{
    die "mk{nod,fifo} $path failed";
}

```

A fifo is convenient when you want to connect a process to an unrelated one. When you open a fifo, the program will block until there's something on the other end.

For example, let's say you'd like to have your *.signature* file be a named pipe that has a Perl program on the other end. Now every time any program (like a mailer, news reader, finger program, etc.) tries to read from that file, the reading program will block and your program will supply the new signature. We'll use the pipe-checking file test `-p` to find out whether anyone (or anything) has accidentally removed our fifo.

```

chdir; # go home
$FIFO = '.signature';
$ENV{PATH} .= ":/etc:/usr/games";

while (1) {
    unless (-p $FIFO) {
        unlink $FIFO;
        system('mknod', $FIFO, 'p')
            && die "can't mknod $FIFO: $!";
    }

    # next line blocks until there's a reader
    open (FIFO, "> $FIFO") || die "can't write $FIFO: $!";
    print FIFO "John Smith (smith@host.org)\n", `fortune -s`;
}

```

```
        close FIFO;
        sleep 2;      # to avoid dup signals
    }
```

Deferred Signals

In Perls before Perl 5.7.3 by installing Perl code to deal with signals, you were exposing yourself to danger from two things. First, few system library functions are re-entrant. If the signal interrupts while Perl is executing one function (like `malloc(3)` or `printf(3)`), and your signal handler then calls the same function again, you could get unpredictable behavior—often, a core dump. Second, Perl isn't itself re-entrant at the lowest levels. If the signal interrupts Perl while Perl is changing its own internal data structures, similarly unpredictable behaviour may result.

There were two things you could do, knowing this: be paranoid or be pragmatic. The paranoid approach was to do as little as possible in your signal handler. Set an existing integer variable that already has a value, and return. This doesn't help you if you're in a slow system call, which will just restart. That means you have to `die` to `longjump(3)` out of the handler. Even this is a little cavalier for the true paranoiac, who avoids `die` in a handler because the system *is* out to get you. The pragmatic approach was to say "I know the risks, but prefer the convenience", and to do anything you wanted in your signal handler, and be prepared to clean up core dumps now and again.

In Perl 5.7.3 and later to avoid these problems signals are "deferred"—that is when the signal is delivered to the process by the system (to the C code that implements Perl) a flag is set, and the handler returns immediately. Then at strategic "safe" points in the Perl interpreter (e.g. when it is about to execute a new opcode) the flags are checked and the Perl level handler from `%SIG` is executed. The "deferred" scheme allows much more flexibility in the coding of signal handler as we know Perl interpreter is in a safe state, and that we are not in a system library function when the handler is called. However the implementation does differ from previous Perls in the following ways:

Long running opcodes

As Perl interpreter only looks at the signal flags when it about to execute a new opcode if a signal arrives during a long running opcode (e.g. a regular expression operation on a very large string) then signal will not be seen until operation completes.

Interrupting IO

When a signal is delivered (e.g. `INT` control-C) the operating system breaks into IO operations like `read` (used to implement Perls `<>` operator). On older Perls the handler was called immediately (and as `read` is not "unsafe" this worked well). With the "deferred" scheme the handler is not called immediately, and if Perl is using system's `stdio` library that library may re-start the `read` without returning to Perl and giving it a chance to call the `%SIG` handler. If this happens on your system the solution is to use `:perlio` layer to do IO – at least on those handles which you want to be able to break into with signals. (The `:perlio` layer checks the signal flags and calls `%SIG` handlers before resuming IO operation.)

Note that the default in Perl 5.7.3 and later is to automatically use the `:perlio` layer.

Signals as "faults"

Certain signals e.g. `SEGV`, `ILL`, `BUS` are generated as a result of virtual memory or other "faults". These are normally fatal and there is little a Perl-level handler can do with them. (In particular the old signal scheme was particularly unsafe in such cases.) However if a `%SIG` handler is set the new scheme simply sets a flag and returns as described above. This may cause the operating system to try the offending machine instruction again and – as nothing has changed – it will generate the signal again. The result of this is a rather odd "loop". In future Perl's signal mechanism may be changed to avoid this – perhaps by simply disallowing `%SIG` handlers on signals of that type. Until then the work-around is not to set a `%SIG` handler on those signals. (Which signals they are is operating system dependant.)

Signals triggered by operating system state

On some operating systems certain signal handlers are supposed to "do something" before returning. One example can be CHLD or CLD which indicates a child process has completed. On some operating systems the signal handler is expected to wait for the completed child process. On such systems the deferred signal scheme will not work for those signals (it does not do the wait). Again the failure will look like a loop as the operating system will re-issue the signal as there are un-waited-for completed child processes.

Using open () for IPC

Perl's basic open () statement can also be used for unidirectional interprocess communication by either appending or prepending a pipe symbol to the second argument to open (). Here's how to start something up in a child process you intend to write to:

```
open(SPOOLER, "| cat -v | lpr -h 2>/dev/null")
    || die "can't fork: $!";
local $SIG{PIPE} = sub { die "spooler pipe broke" };
print SPOOLER "stuff\n";
close SPOOLER || die "bad spool: $! $?";
```

And here's how to start up a child process you intend to read from:

```
open(STATUS, "netstat -an 2>&1 |")
    || die "can't fork: $!";
while (<STATUS>) {
    next if /^(tcp|udp)/;
    print;
}
close STATUS || die "bad netstat: $! $?";
```

If one can be sure that a particular program is a Perl script that is expecting filenames in @ARGV, the clever programmer can write something like this:

```
% program f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

and irrespective of which shell it's called from, the Perl program will read from the file *f1*, the process *cmd1*, standard input (*tmpfile* in this case), the *f2* file, the *cmd2* command, and finally the *f3* file. Pretty nifty, eh?

You might notice that you could use backticks for much the same effect as opening a pipe for reading:

```
print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
die "bad netstat" if $?;
```

While this is true on the surface, it's much more efficient to process the file one line or record at a time because then you don't have to read the whole thing into memory at once. It also gives you finer control of the whole process, letting you to kill off the child process early if you'd like.

Be careful to check both the open () and the close () return values. If you're *writing* to a pipe, you should also trap SIGPIPE. Otherwise, think of what happens when you start up a pipe to a command that doesn't exist: the open () will in all likelihood succeed (it only reflects the fork ()'s success), but then your output will fail—spectacularly. Perl can't know whether the command worked because your command is actually running in a separate process whose exec () might have failed. Therefore, while readers of bogus commands return just a quick end of file, writers to bogus command will trigger a signal they'd better be prepared to handle. Consider:

```
open(FH, "|bogus")    or die "can't fork: $!";
print FH "bang\n"     or die "can't write: $!";
close FH              or die "can't close: $!";
```

That won't blow up until the close, and it will blow up with a SIGPIPE. To catch it, you could use this:

```
$SIG{PIPE} = 'IGNORE';
open(FH, "|bogus") or die "can't fork: $!";
print FH "bang\n" or die "can't write: $!";
close FH or die "can't close: status=$?";
```

Filehandles

Both the main process and any child processes it forks share the same STDIN, STDOUT, and STDERR filehandles. If both processes try to access them at once, strange things can happen. You may also want to close or reopen the filehandles for the child. You can get around this by opening your pipe with `open()`, but on some systems this means that the child process cannot outlive the parent.

Background Processes

You can run a command in the background with:

```
system("cmd &");
```

The command's STDOUT and STDERR (and possibly STDIN, depending on your shell) will be the same as the parent's. You won't need to catch SIGCHLD because of the double-fork taking place (see below for more details).

Complete Dissociation of Child from Parent

In some cases (starting server processes, for instance) you'll want to completely dissociate the child process from the parent. This is often called daemonization. A well behaved daemon will also `chdir()` to the root directory (so it doesn't prevent unmounting the filesystem containing the directory from which it was launched) and redirect its standard file descriptors from and to */dev/null* (so that random output doesn't wind up on the user's terminal).

```
use POSIX 'setsid';

sub daemonize {
    chdir '/' or die "Can't chdir to /: $!";
    open STDIN, '/dev/null' or die "Can't read /dev/null: $!";
    open STDOUT, '>/dev/null' or die "Can't write to /dev/null: $!";
    defined(my $pid = fork) or die "Can't fork: $!";
    exit if $pid;
    setsid or die "Can't start a new session: $!";
    open STDERR, '>&STDOUT' or die "Can't dup stdout: $!";
}
```

The `fork()` has to come before the `setsid()` to ensure that you aren't a process group leader (the `setsid()` will fail if you are). If your system doesn't have the `setsid()` function, open */dev/tty* and use the `TIOCNOTTY` `ioctl()` on it instead. See [tty\(4\)](#) for details.

Non-Unix users should check their `Your_OS::Process` module for other solutions.

Safe Pipe Opens

Another interesting approach to IPC is making your single program go multiprocess and communicate between (or even amongst) yourselves. The `open()` function will accept a file argument of either `"-|"` or `"|-"` to do a very interesting thing: it forks a child connected to the filehandle you've opened. The child is running the same program as the parent. This is useful for safely opening a file when running under an assumed UID or GID, for example. If you open a pipe *to* minus, you can write to the filehandle you opened and your kid will find it in his STDIN. If you open a pipe *from* minus, you can read from the filehandle you opened whatever your kid writes to his STDOUT.

```
use English '-no_match_vars';
my $sleep_count = 0;

do {
    $pid = open(KID_TO_WRITE, "|-");
```

```

        unless (defined $pid) {
            warn "cannot fork: $!";
            die "bailing out" if $sleep_count++ > 6;
            sleep 10;
        }
    } until defined $pid;
    if ($pid) { # parent
        print KID_TO_WRITE @some_data;
        close(KID_TO_WRITE) || warn "kid exited $?";
    } else { # child
        ($EUID, $EGID) = ($UID, $GID); # suid progs only
        open (FILE, "> /safe/file")
            || die "can't open /safe/file: $!";
        while (<STDIN>) {
            print FILE; # child's STDIN is parent's KID
        }
        exit; # don't forget this
    }
}

```

Another common use for this construct is when you need to execute something without the shell's interference. With `system()`, it's straightforward, but you can't use a pipe open or backticks safely. That's because there's no way to stop the shell from getting its hands on your arguments. Instead, use lower-level control to call `exec()` directly.

Here's a safe backtick or pipe open for read:

```

# add error processing as above
$pid = open(KID_TO_READ, "-|");

if ($pid) { # parent
    while (<KID_TO_READ>) {
        # do something interesting
    }
    close(KID_TO_READ) || warn "kid exited $?";
} else { # child
    ($EUID, $EGID) = ($UID, $GID); # suid only
    exec($program, @options, @args)
        || die "can't exec program: $!";
    # NOTREACHED
}

```

And here's a safe pipe open for writing:

```

# add error processing as above
$pid = open(KID_TO_WRITE, "|-");
$SIG{ALRM} = sub { die "whoops, $program pipe broke" };

if ($pid) { # parent
    for (@data) {
        print KID_TO_WRITE;
    }
    close(KID_TO_WRITE) || warn "kid exited $?";
} else { # child
    ($EUID, $EGID) = ($UID, $GID);
    exec($program, @options, @args)
        || die "can't exec program: $!";
}

```

```
    # NOTREACHED
}
```

Since Perl 5.8.0, you can also use the list form of `open` for pipes : the syntax

```
open KID_PS, "-|", "ps", "aux" or die $!;
```

forks the `ps(1)` command (without spawning a shell, as there are more than three arguments to `open()`), and reads its standard output via the `KID_PS` filehandle. The corresponding syntax to read from command pipes (with `"|"` in place of `"-|"`) is also implemented.

Note that these operations are full Unix forks, which means they may not be correctly implemented on alien systems. Additionally, these are not true multithreading. If you'd like to learn more about threading, see the *modules* file mentioned below in the SEE ALSO section.

Bidirectional Communication with Another Process

While this works reasonably well for unidirectional communication, what about bidirectional communication? The obvious thing you'd like to do doesn't actually work:

```
open(PROG_FOR_READING_AND_WRITING, "| some program |")
```

and if you forget to use the `use warnings` pragma or the `-w` flag, then you'll miss out entirely on the diagnostic message:

```
Can't do bidirectional pipe at -e line 1.
```

If you really want to, you can use the standard `open2()` library function to catch both ends. There's also an `open3()` for tridirectional I/O so you can also catch your child's `STDERR`, but doing so would then require an awkward `select()` loop and wouldn't allow you to use normal Perl input operations.

If you look at its source, you'll see that `open2()` uses low-level primitives like Unix `pipe()` and `exec()` calls to create all the connections. While it might have been slightly more efficient by using `socketpair()`, it would have then been even less portable than it already is. The `open2()` and `open3()` functions are unlikely to work anywhere except on a Unix system or some other one purporting to be POSIX compliant.

Here's an example of using `open2()`:

```
use FileHandle;
use IPC::Open2;
$pid = open2(*Reader, *Writer, "cat -u -n" );
print Writer "stuff\n";
$got = <Reader>;
```

The problem with this is that Unix buffering is really going to ruin your day. Even though your `Writer` filehandle is auto-flushed, and the process on the other end will get your data in a timely manner, you can't usually do anything to force it to give it back to you in a similarly quick fashion. In this case, we could, because we gave `cat` a `-u` flag to make it unbuffered. But very few Unix commands are designed to operate over pipes, so this seldom works unless you yourself wrote the program on the other end of the double-ended pipe.

A solution to this is the nonstandard *Comm.pl* library. It uses pseudo-ttys to make your program behave more reasonably:

```
require 'Comm.pl';
$ph = open_proc('cat -n');
for (1..10) {
    print $ph "a line\n";
    print "got back ", scalar <$ph>;
}
```

This way you don't have to have control over the source code of the program you're using. The *Comm*

library also has `expect()` and `interact()` functions. Find the library (and we hope its successor *IPC::Chat*) at your nearest CPAN archive as detailed in the SEE ALSO section below.

The newer `Expect.pm` module from CPAN also addresses this kind of thing. This module requires two other modules from CPAN: `IO::Pty` and `IO::Stty`. It sets up a pseudo-terminal to interact with programs that insist on using talking to the terminal device driver. If your system is amongst those supported, this may be your best bet.

Bidirectional Communication with Yourself

If you want, you may make low-level `pipe()` and `fork()` to stitch this together by hand. This example only talks to itself, but you could reopen the appropriate handles to `STDIN` and `STDOUT` and call other processes.

```
#!/usr/bin/perl -w
# pipe1 - bidirectional communication using two pipe pairs
#       designed for the socketpair-challenged
use IO::Handle;      # thousands of lines just for autoflush :-(
pipe(PARENT_RDR, CHILD_WTR);      # XXX: failure?
pipe(CHILD_RDR, PARENT_WTR);      # XXX: failure?
CHILD_WTR->autoflush(1);
PARENT_WTR->autoflush(1);

if ($pid = fork) {
    close PARENT_RDR; close PARENT_WTR;
    print CHILD_WTR "Parent Pid $$ is sending this\n";
    chomp($line = <CHILD_RDR>);
    print "Parent Pid $$ just read this: '$line'\n";
    close CHILD_RDR; close CHILD_WTR;
    waitpid($pid, 0);
} else {
    die "cannot fork: $!" unless defined $pid;
    close CHILD_RDR; close CHILD_WTR;
    chomp($line = <PARENT_RDR>);
    print "Child Pid $$ just read this: '$line'\n";
    print PARENT_WTR "Child Pid $$ is sending this\n";
    close PARENT_RDR; close PARENT_WTR;
    exit;
}
```

But you don't actually have to make two pipe calls. If you have the `socketpair()` system call, it will do this all for you.

```
#!/usr/bin/perl -w
# pipe2 - bidirectional communication using socketpair
#       "the best ones always go both ways"

use Socket;
use IO::Handle;      # thousands of lines just for autoflush :-(
# We say AF_UNIX because although *_LOCAL is the
# POSIX 1003.1g form of the constant, many machines
# still don't have it.
socketpair(CHILD, PARENT, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "socketpair: $!";

CHILD->autoflush(1);
PARENT->autoflush(1);

if ($pid = fork) {
    close PARENT;
```

```

        print CHILD "Parent Pid $$ is sending this\n";
        chomp($line = <CHILD>);
        print "Parent Pid $$ just read this: '$line'\n";
        close CHILD;
        waitpid($pid,0);
    } else {
        die "cannot fork: $!" unless defined $pid;
        close CHILD;
        chomp($line = <PARENT>);
        print "Child Pid $$ just read this: '$line'\n";
        print PARENT "Child Pid $$ is sending this\n";
        close PARENT;
        exit;
    }
}

```

Sockets: Client/Server Communication

While not limited to Unix-derived operating systems (e.g., WinSock on PCs provides socket support, as do some VMS libraries), you may not have sockets on your system, in which case this section probably isn't going to do you much good. With sockets, you can do both virtual circuits (i.e., TCP streams) and datagrams (i.e., UDP packets). You may be able to do even more depending on your system.

The Perl function calls for dealing with sockets have the same names as the corresponding system calls in C, but their arguments tend to differ for two reasons: first, Perl filehandles work differently than C file descriptors. Second, Perl already knows the length of its strings, so you don't need to pass that information.

One of the major problems with old socket code in Perl was that it used hard-coded values for some of the constants, which severely hurt portability. If you ever see code that does anything like explicitly setting `$AF_INET = 2`, you know you're in for big trouble: An immeasurably superior approach is to use the `Socket` module, which more reliably grants access to various constants and functions you'll need.

If you're not writing a server/client for an existing protocol like NNTP or SMTP, you should give some thought to how your server will know when the client has finished talking, and vice-versa. Most protocols are based on one-line messages and responses (so one party knows the other has finished when a "\n" is received) or multi-line messages and responses that end with a period on an empty line ("\n.\n" terminates a message/response).

Internet Line Terminators

The Internet line terminator is "\015\012". Under ASCII variants of Unix, that could usually be written as "\r\n", but under other systems, "\r\n" might at times be "\015\015\012", "\012\012\015", or something completely different. The standards specify writing "\015\012" to be conformant (be strict in what you provide), but they also recommend accepting a lone "\012" on input (but be lenient in what you require). We haven't always been very good about that in the code in this manpage, but unless you're on a Mac, you'll probably be ok.

Internet TCP Clients and Servers

Use Internet-domain sockets when you want to do client-server communication that might extend to machines outside of your own system.

Here's a sample TCP client using Internet-domain sockets:

```

#!/usr/bin/perl -w
use strict;
use Socket;
my ($remote,$port, $iaddr, $paddr, $proto, $line);

$remote = shift || 'localhost';
$port   = shift || 2345; # random port
if ($port =~ /\D/) { $port = getservbyname($port, 'tcp') }

```

```

die "No port" unless $port;
$iaddr = inet_aton($remote) || die "no host: $remote";
$paddr = sockaddr_in($port, $iaddr);

$proto = getprotobyname('tcp');
socket(SOCK, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
connect(SOCK, $paddr) || die "connect: $!";
while (defined($line = <SOCK>)) {
    print $line;
}

close (SOCK) || die "close: $!";
exit;

```

And here's a corresponding server to go along with it. We'll leave the address as INADDR_ANY so that the kernel can choose the appropriate interface on multihomed hosts. If you want sit on a particular interface (like the external side of a gateway or firewall machine), you should fill this in with your real address instead.

```

#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;
my $EOL = "\015\012";

sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

my $port = shift || 2345;
my $proto = getprotobyname('tcp');

($port) = $port =~ /^(\d+)$/ or die "invalid port";

socket(Server, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
            pack("l", 1)) || die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY)) || die "bind: $!";
listen(Server, SOMAXCONN) || die "listen: $!";

logmsg "server started on port $port";

my $paddr;
$SIG{CHLD} = \&REAPER;

for ( ; $paddr = accept(Client, Server); close Client) {
    my($port, $iaddr) = sockaddr_in($paddr);
    my $name = gethostbyaddr($iaddr, AF_INET);

    logmsg "connection from $name [",
           inet_ntoa($iaddr), "]"
           at port $port";

    print Client "Hello there, $name, it's now ",
               scalar localtime, $EOL;
}

```

And here's a multithreaded version. It's multithreaded in that like most typical servers, it spawns (forks) a slave server to handle the client request so that the master server can quickly go back to service a new client.

```

#!/usr/bin/perl -Tw
use strict;

```

```

BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;
my $EOL = "\015\012";

sub spawn; # forward declaration
sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

my $port = shift || 2345;
my $proto = getprotobyname('tcp');

($port) = $port =~ /^(\d+)$/ or die "invalid port";

socket(Server, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
            pack("l", 1)) || die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY)) || die "bind: $!";
listen(Server, SOMAXCONN) || die "listen: $!";

logmsg "server started on port $port";

my $waitedpid = 0;
my $paddr;

use POSIX ":sys_wait_h";
sub REAPER {
    my $child;
    while (($waitedpid = waitpid(-1, WNOHANG)) > 0) {
        logmsg "reaped $waitedpid" . ($? ? " with exit $" : '');
    }
    $SIG{CHLD} = \&REAPER; # loathe sysV
}

$SIG{CHLD} = \&REAPER;

for ( $waitedpid = 0;
      ($paddr = accept(Client, Server)) || $waitedpid;
      $waitedpid = 0, close Client)
{
    next if $waitedpid and not $paddr;
    my($port, $iaddr) = sockaddr_in($paddr);
    my $name = gethostbyaddr($iaddr, AF_INET);

    logmsg "connection from $name [",
           inet_ntoa($iaddr), "]"
           at port $port";

    spawn sub {
        $|=1;
        print "Hello there, $name, it's now ", scalar localtime, $EOL;
        exec '/usr/games/fortune' # XXX: 'wrong' line terminators
        or confess "can't exec fortune: $!";
    };
}

sub spawn {
    my $coderef = shift;

    unless (@_ == 0 && $coderef && ref($coderef) eq 'CODE') {
        confess "usage: spawn CODEREF";
    }
}

```

```

    }
    my $pid;
    if (!defined($pid = fork)) {
        logmsg "cannot fork: $!";
        return;
    } elsif ($pid) {
        logmsg "begat $pid";
        return; # I'm the parent
    }
    # else I'm the child -- go spawn

    open(STDIN, "<&Client") || die "can't dup client to stdin";
    open(STDOUT, ">&Client") || die "can't dup client to stdout";
    ## open(STDERR, ">&STDOUT") || die "can't dup stdout to stderr";
    exit &$coderef();
}

```

This server takes the trouble to clone off a child version via `fork()` for each incoming request. That way it can handle many requests at once, which you might not always want. Even if you don't `fork()`, the `listen()` will allow that many pending connections. Forking servers have to be particularly careful about cleaning up their dead children (called "zombies" in Unix parlance), because otherwise you'll quickly fill up your process table.

We suggest that you use the `-T` flag to use taint checking (see [perlsec](#)) even if we aren't running `setuid` or `setgid`. This is always a good idea for servers and other programs run on behalf of someone else (like CGI scripts), because it lessens the chances that people from the outside will be able to compromise your system.

Let's look at another TCP client. This one connects to the TCP "time" service on a number of different machines and shows how far their clocks differ from the system on which it's being run:

```

#!/usr/bin/perl -w
use strict;
use Socket;

my $SECS_of_70_YEARS = 2208988800;
sub ctime { scalar localtime(shift) }

my $iaddr = gethostbyname('localhost');
my $proto = getprotobyname('tcp');
my $port = getservbyname('time', 'tcp');
my $paddr = sockaddr_in(0, $iaddr);
my($host);

$| = 1;
printf "%-24s %8s %s\n", "localhost", 0, ctime(time());

foreach $host (@ARGV) {
    printf "%-24s ", $host;
    my $hisiaddr = inet_aton($host) || die "unknown host";
    my $hispaddr = sockaddr_in($port, $hisiaddr);
    socket(SOCKET, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
    connect(SOCKET, $hispaddr) || die "bind: $!";
    my $rtime = ' ';
    read(SOCKET, $rtime, 4);
    close(SOCKET);
    my $histime = unpack("N", $rtime) - $SECS_of_70_YEARS;
    printf "%8d %s\n", $histime - time, ctime($histime);
}

```

Unix-Domain TCP Clients and Servers

That's fine for Internet-domain clients and servers, but what about local communications? While you can use the same setup, sometimes you don't want to. Unix-domain sockets are local to the current host, and are often used internally to implement pipes. Unlike Internet domain sockets, Unix domain sockets can show up in the file system with an `ls(1)` listing.

```
% ls -l /dev/log
srw-rw-rw-  1 root          0 Oct 31 07:23 /dev/log
```

You can test for these with Perl's `-S` file test:

```
unless ( -S '/dev/log' ) {
    die "something's wicked with the log system";
}
```

Here's a sample Unix-domain client:

```
#!/usr/bin/perl -w
use Socket;
use strict;
my ($rendezvous, $line);

$rendezvous = shift || '/tmp/catsock';
socket(SOCK, PF_UNIX, SOCK_STREAM, 0)      || die "socket: $!";
connect(SOCK, sockaddr_un($rendezvous))    || die "connect: $!";
while (defined($line = <SOCK>)) {
    print $line;
}
exit;
```

And here's a corresponding server. You don't have to worry about silly network terminators here because Unix domain sockets are guaranteed to be on the localhost, and thus everything works right.

```
#!/usr/bin/perl -Tw
use strict;
use Socket;
use Carp;

BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
sub spawn; # forward declaration
sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

my $NAME = '/tmp/catsock';
my $uaddr = sockaddr_un($NAME);
my $proto = getprotobyname('tcp');

socket(Server, PF_UNIX, SOCK_STREAM, 0)      || die "socket: $!";
unlink($NAME);
bind (Server, $uaddr)                        || die "bind: $!";
listen(Server, SOMAXCONN)                   || die "listen: $!";

logmsg "server started on $NAME";

my $waitedpid;

use POSIX ":sys_wait_h";
sub REAPER {
    my $child;
    while (($waitedpid = waitpid(-1, WNOHANG)) > 0) {
        logmsg "reaped $waitedpid" . ($? ? " with exit $" : '');
    }
}
```

```

        $SIG{CHLD} = \&REAPER; # loathe sysV
    }

    $SIG{CHLD} = \&REAPER;
    for ( $waitedpid = 0;
        accept(Client,Server) || $waitedpid;
        $waitedpid = 0, close Client)
    {
        next if $waitedpid;
        logmsg "connection on $NAME";
        spawn sub {
            print "Hello there, it's now ", scalar localtime, "\n";
            exec '/usr/games/fortune' or die "can't exec fortune: $!";
        };
    }

    sub spawn {
        my $coderef = shift;

        unless (@_ == 0 && $coderef && ref($coderef) eq 'CODE') {
            confess "usage: spawn CODEREF";
        }

        my $pid;
        if (!defined($pid = fork)) {
            logmsg "cannot fork: $!";
            return;
        } elsif ($pid) {
            logmsg "begat $pid";
            return; # I'm the parent
        }
        # else I'm the child -- go spawn

        open(STDIN, "<&Client") || die "can't dup client to stdin";
        open(STDOUT, ">&Client") || die "can't dup client to stdout";
        ## open(STDERR, ">&STDOUT") || die "can't dup stdout to stderr";
        exit &$coderef();
    }

```

As you see, it's remarkably similar to the Internet domain TCP server, so much so, in fact, that we've omitted several duplicate functions—`spawn()`, `logmsg()`, `ctime()`, and `REAPER()`—which are exactly the same as in the other server.

So why would you ever want to use a Unix domain socket instead of a simpler named pipe? Because a named pipe doesn't give you sessions. You can't tell one process's data from another's. With socket programming, you get a separate session for each client: that's why `accept()` takes two arguments.

For example, let's say that you have a long running database server daemon that you want folks from the World Wide Web to be able to access, but only if they go through a CGI interface. You'd have a small, simple CGI program that does whatever checks and logging you feel like, and then acts as a Unix-domain client and connects to your private server.

TCP Clients with IO::Socket

For those preferring a higher-level interface to socket programming, the `IO::Socket` module provides an object-oriented approach. `IO::Socket` is included as part of the standard Perl distribution as of the 5.004 release. If you're running an earlier version of Perl, just fetch `IO::Socket` from CPAN, where you'll also find modules providing easy interfaces to the following systems: DNS, FTP, Ident (RFC 931), NIS and NISPlus, NNTP, Ping, POP3, SMTP, SNMP, SSLey, Telnet, and Time—just to name a few.

A Simple Client

Here's a client that creates a TCP connection to the "daytime" service at port 13 of the host name "localhost" and prints out everything that the server there cares to provide.

```
#!/usr/bin/perl -w
use IO::Socket;
$remote = IO::Socket::INET->new(
    Proto      => "tcp",
    PeerAddr   => "localhost",
    PeerPort   => "daytime(13)",
)
    or die "cannot connect to daytime port at localhost";
while ( <$remote> ) { print }
```

When you run this program, you should get something back that looks like this:

```
Wed May 14 08:40:46 MDT 1997
```

Here are what those parameters to the new constructor mean:

Proto

This is which protocol to use. In this case, the socket handle returned will be connected to a TCP socket, because we want a stream-oriented connection, that is, one that acts pretty much like a plain old file. Not all sockets are this of this type. For example, the UDP protocol can be used to make a datagram socket, used for message-passing.

PeerAddr

This is the name or Internet address of the remote host the server is running on. We could have specified a longer name like "www.perl.com", or an address like "204.148.40.9". For demonstration purposes, we've used the special hostname "localhost", which should always mean the current machine you're running on. The corresponding Internet address for localhost is "127.1", if you'd rather use that.

PeerPort

This is the service name or port number we'd like to connect to. We could have gotten away with using just "daytime" on systems with a well-configured system services file,[FOOTNOTE: The system services file is in */etc/services* under Unix] but just in case, we've specified the port number (13) in parentheses. Using just the number would also have worked, but constant numbers make careful programmers nervous.

Notice how the return value from the new constructor is used as a filehandle in the while loop? That's what's called an indirect filehandle, a scalar variable containing a filehandle. You can use it the same way you would a normal filehandle. For example, you can read one line from it this way:

```
$line = <$handle>;
```

all remaining lines from is this way:

```
@lines = <$handle>;
```

and send a line of data to it this way:

```
print $handle "some data\n";
```

A Webget Client

Here's a simple client that takes a remote host to fetch a document from, and then a list of documents to get from that host. This is a more interesting client than the previous one because it first sends something to the server before fetching the server's response.

```
#!/usr/bin/perl -w
use IO::Socket;
```



```

unless (@ARGV > 1) { die "usage: $0 host document ..." }
$host = shift(@ARGV);
$EOL = "\015\012";
$BLANK = $EOL x 2;
foreach $document ( @ARGV ) {
    $remote = IO::Socket::INET->new( Proto      => "tcp",
                                     PeerAddr   => $host,
                                     PeerPort   => "http(80)",
                                     );
    unless ($remote) { die "cannot connect to http daemon on $host" }
    $remote->autoflush(1);
    print $remote "GET $document HTTP/1.0" . $BLANK;
    while ( <$remote> ) { print }
    close $remote;
}

```

The web server handling the "http" service, which is assumed to be at its standard port, number 80. If the web server you're trying to connect to is at a different port (like 1080 or 8080), you should specify as the named-parameter pair, `< PeerPort = 8080`. The `autoflush` method is used on the socket because otherwise the system would buffer up the output we sent it. (If you're on a Mac, you'll also need to change every `"\n"` in your code that sends data over the network to be a `"\015\012"` instead.)

Connecting to the server is only the first part of the process: once you have the connection, you have to use the server's language. Each server on the network has its own little command language that it expects as input. The string that we send to the server starting with "GET" is in HTTP syntax. In this case, we simply request each specified document. Yes, we really are making a new connection for each document, even though it's the same host. That's the way you always used to have to speak HTTP. Recent versions of web browsers may request that the remote server leave the connection open a little while, but the server doesn't have to honor such a request.

Here's an example of running that program, which we'll call *webget*:

```

% webget www.perl.com /guanaco.html
HTTP/1.1 404 File Not Found
Date: Thu, 08 May 1997 18:02:32 GMT
Server: Apache/1.2b6
Connection: close
Content-type: text/html

<HEAD><TITLE>404 File Not Found</TITLE></HEAD>
<BODY><H1>File Not Found</H1>
The requested URL /guanaco.html was not found on this server.<P>
</BODY>

```

Ok, so that's not very interesting, because it didn't find that particular document. But a long response wouldn't have fit on this page.

For a more fully-featured version of this program, you should look to the *lwp-request* program included with the LWP modules from CPAN.

Interactive Client with IO::Socket

Well, that's all fine if you want to send one command and get one answer, but what about setting up something fully interactive, somewhat like the way *telnet* works? That way you can type a line, get the answer, type a line, get the answer, etc.

This client is more complicated than the two we've done so far, but if you're on a system that supports the powerful `fork` call, the solution isn't that rough. Once you've made the connection to whatever service you'd like to chat with, call `fork` to clone your process. Each of these two identical process has a very simple job to do: the parent copies everything from the socket to standard output, while the child

simultaneously copies everything from standard input to the socket. To accomplish the same thing using just one process would be *much* harder, because it's easier to code two processes to do one thing than it is to code one process to do two things. (This keep-it-simple principle a cornerstones of the Unix philosophy, and good software engineering as well, which is probably why it's spread to other systems.)

Here's the code:

```
#!/usr/bin/perl -w
use strict;
use IO::Socket;
my ($host, $port, $kidpid, $handle, $line);

unless (@ARGV == 2) { die "usage: $0 host port" }
($host, $port) = @ARGV;

# create a tcp connection to the specified host and port
$handle = IO::Socket::INET->new(Proto    => "tcp",
                               PeerAddr  => $host,
                               PeerPort  => $port)
    or die "can't connect to port $port on $host: $!";

$handle->autoflush(1);          # so output gets there right away
print STDERR "[Connected to $host:$port]\n";

# split the program into two processes, identical twins
die "can't fork: $!" unless defined($kidpid = fork());

# the if{} block runs only in the parent process
if ($kidpid) {
    # copy the socket to standard output
    while (defined ($line = <$handle>)) {
        print STDOUT $line;
    }
    kill("TERM", $kidpid);      # send SIGTERM to child
}
# the else{} block runs only in the child process
else {
    # copy standard input to the socket
    while (defined ($line = <STDIN>)) {
        print $handle $line;
    }
}
```

The `kill` function in the parent's `if` block is there to send a signal to our child process (current running in the `else` block) as soon as the remote server has closed its end of the connection.

If the remote server sends data a byte at time, and you need that data immediately without waiting for a newline (which might not happen), you may wish to replace the `while` loop in the parent with the following:

```
my $byte;
while (sysread($handle, $byte, 1) == 1) {
    print STDOUT $byte;
}
```

Making a system call for each byte you want to read is not very efficient (to put it mildly) but is the simplest to explain and works reasonably well.

TCP Servers with IO::Socket

As always, setting up a server is little bit more involved than running a client. The model is that the server creates a special kind of socket that does nothing but listen on a particular port for incoming connections. It does this by calling the `< IO::Socket::INET->new()` method with slightly different arguments than the client did.

Proto

This is which protocol to use. Like our clients, we'll still specify "tcp" here.

LocalPort

We specify a local port in the `LocalPort` argument, which we didn't do for the client. This is service name or port number for which you want to be the server. (Under Unix, ports under 1024 are restricted to the superuser.) In our sample, we'll use port 9000, but you can use any port that's not currently in use on your system. If you try to use one already in used, you'll get an "Address already in use" message. Under Unix, the `netstat -a` command will show which services current have servers.

Listen

The `Listen` parameter is set to the maximum number of pending connections we can accept until we turn away incoming clients. Think of it as a call-waiting queue for your telephone. The low-level `Socket` module has a special symbol for the system maximum, which is `SOMAXCONN`.

Reuse

The `Reuse` parameter is needed so that we restart our server manually without waiting a few minutes to allow system buffers to clear out.

Once the generic server socket has been created using the parameters listed above, the server then waits for a new client to connect to it. The server blocks in the `accept` method, which eventually accepts a bidirectional connection from the remote client. (Make sure to autoflush this handle to circumvent buffering.)

To add to user-friendliness, our server prompts the user for commands. Most servers don't do this. Because of the prompt without a newline, you'll have to use the `sysread` variant of the interactive client above.

This server accepts one of five different commands, sending output back to the client. Note that unlike most network servers, this one only handles one incoming client at a time. Multithreaded servers are covered in Chapter 6 of the Camel.

Here's the code. We'll

```
#!/usr/bin/perl -w
use IO::Socket;
use Net::hostent;           # for OO version of gethostbyaddr

$PORT = 9000;              # pick something not in use

$server = IO::Socket::INET->new( Proto    => 'tcp',
                                LocalPort => $PORT,
                                Listen    => SOMAXCONN,
                                Reuse     => 1);

die "can't setup server" unless $server;
print "[Server $0 accepting clients]\n";

while ($client = $server->accept()) {
    $client->autoflush(1);
    print $client "Welcome to $0; type help for command list.\n";
    $hostinfo = gethostbyaddr($client->peeraddr);
    printf "[Connect from %s]\n", $hostinfo->name || $client->peerhost;
    print $client "Command? ";
    while ( <$client> ) {
```

```

next unless /\S/ # blank line
if      (/quit|exit/i)    { last; }
elsif  (/date|time/i)    { printf $client "%s\n", scalar localtime; }
elsif  (/who/i)          { print  $client `who 2>&1`; }
elsif  (/cookie/i)       { print  $client `/usr/games/fortune 2>&1`; }
elsif  (/motd/i)         { print  $client `cat /etc/motd 2>&1`; }
else {
    print $client "Commands: quit date who cookie motd\n";
}
} continue {
    print $client "Command? ";
}
close $client;
}

```

UDP: Message Passing

Another kind of client-server setup is one that uses not connections, but messages. UDP communications involve much lower overhead but also provide less reliability, as there are no promises that messages will arrive at all, let alone in order and unmangled. Still, UDP offers some advantages over TCP, including being able to "broadcast" or "multicast" to a whole bunch of destination hosts at once (usually on your local subnet). If you find yourself overly concerned about reliability and start building checks into your message system, then you probably should use just TCP to start with.

Note that UDP datagrams are *not* a bytestream and should not be treated as such. This makes using I/O mechanisms with internal buffering like `stdio` (i.e. `print()` and friends) especially cumbersome. Use `syswrite()`, or better `send()`, like in the example below.

Here's a UDP program similar to the sample Internet TCP client given earlier. However, instead of checking one host at a time, the UDP version will check many of them asynchronously by simulating a multicast and then using `select()` to do a timed-out wait for I/O. To do something similar with TCP, you'd have to use a different socket handle for each host.

```

#!/usr/bin/perl -w
use strict;
use Socket;
use Sys::Hostname;

my ( $count, $hisiaddr, $hispaddr, $histime,
     $host, $iaddr, $paddr, $port, $proto,
     $rin, $rout, $rtime, $SECS_of_70_YEARS);

$SECS_of_70_YEARS      = 2208988800;

$iaddr = gethostbyname(hostname());
$proto = getprotobyname('udp');
$port  = getservbyname('time', 'udp');
$paddr = sockaddr_in(0, $iaddr); # 0 means let kernel pick

socket(SOCKET, PF_INET, SOCK_DGRAM, $proto) || die "socket: $!";
bind(SOCKET, $paddr)                        || die "bind: $!";

$| = 1;
printf "%-12s %8s %s\n", "localhost", 0, scalar localtime time;
$count = 0;
for $host (@ARGV) {
    $count++;
    $hisiaddr = inet_aton($host) || die "unknown host";
    $hispaddr = sockaddr_in($port, $hisiaddr);
    defined(send(SOCKET, 0, 0, $hispaddr)) || die "send $host: $!";
}

```

```

    }

    $rin = '';
    vec($rin, fileno(SOCKET), 1) = 1;

    # timeout after 10.0 seconds
    while ($count && select($rout = $rin, undef, undef, 10.0)) {
        $rtime = '';
        ($hispaddr = recv(SOCKET, $rtime, 4, 0)) || die "recv: $!";
        ($port, $hisiaddr) = sockaddr_in($hispaddr);
        $host = gethostbyaddr($hisiaddr, AF_INET);
        $histime = unpack("N", $rtime) - $SECS_of_70_YEARS ;
        printf "%-12s ", $host;
        printf "%8d %s\n", $histime - time, scalar localtime($histime);
        $count--;
    }

```

Note that this example does not include any retries and may consequently fail to contact a reachable host. The most prominent reason for this is congestion of the queues on the sending host if the number of list of hosts to contact is sufficiently large.

SysV IPC

While System V IPC isn't so widely used as sockets, it still has some interesting uses. You can't, however, effectively use SysV IPC or Berkeley mmap() to have shared memory so as to share a variable amongst several processes. That's because Perl would reallocate your string when you weren't wanting it to.

Here's a small example showing shared memory usage.

```

use IPC::SysV qw(IPC_PRIVATE IPC_RMID S_IRWXU);

$size = 2000;
$id = shmget(IPC_PRIVATE, $size, S_IRWXU) || die "$!";
print "shm key $id\n";

$message = "Message #1";
shmwrite($id, $message, 0, 60) || die "$!";
print "wrote: '$message'\n";
shmread($id, $buff, 0, 60) || die "$!";
print "read : '$buff'\n";

# the buffer of shmread is zero-character end-padded.
substr($buff, index($buff, "\0")) = '';
print "un" unless $buff eq $message;
print "swell\n";

print "deleting shm $id\n";
shmctl($id, IPC_RMID, 0) || die "$!";

```

Here's an example of a semaphore:

```

use IPC::SysV qw(IPC_CREAT);

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 10, 0666 | IPC_CREAT) || die "$!";
print "shm key $id\n";

```

Put this code in a separate file to be run in more than one process. Call the file *take*:

```

# create a semaphore

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 0, 0);

```

```

die if !defined($id);

$semnum = 0;
$semflag = 0;

# 'take' semaphore
# wait for semaphore to be zero
$semop = 0;
$opstring1 = pack("s!s!s!", $semnum, $semop, $semflag);

# Increment the semaphore count
$semop = 1;
$opstring2 = pack("s!s!s!", $semnum, $semop, $semflag);
$opstring = $opstring1 . $opstring2;

semop($id,$opstring) || die "$!";

```

Put this code in a separate file to be run in more than one process. Call this file *give*:

```

# 'give' the semaphore
# run this in the original process and you will see
# that the second process continues

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 0, 0);
die if !defined($id);

$semnum = 0;
$semflag = 0;

# Decrement the semaphore count
$semop = -1;
$opstring = pack("s!s!s!", $semnum, $semop, $semflag);

semop($id,$opstring) || die "$!";

```

The SysV IPC code above was written long ago, and it's definitely clunky looking. For a more modern look, see the IPC::SysV module which is included with Perl starting from Perl 5.005.

A small example demonstrating SysV message queues:

```

use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRWXU);

my $id = msgget(IPC_PRIVATE, IPC_CREAT | S_IRWXU);

my $sent = "message";
my $type = 1234;
my $rcvd;
my $type_rcvd;

if (defined $id) {
    if (msgsnd($id, pack("l! a*", $type_sent, $sent), 0)) {
        if (msgrcv($id, $rcvd, 60, 0, 0)) {
            ($type_rcvd, $rcvd) = unpack("l! a*", $rcvd);
            if ($rcvd eq $sent) {
                print "okay\n";
            } else {
                print "not okay\n";
            }
        } else {
            die "# msgrcv failed\n";
        }
    }
}

```

```
    } else {  
        die "# msgsnd failed\n";  
    }  
    msgctl($id, IPC_RMID, 0) || die "# msgctl failed: $!\n";  
} else {  
    die "# msgget failed\n";  
}
```

NOTES

Most of these routines quietly but politely return `undef` when they fail instead of causing your program to die right then and there due to an uncaught exception. (Actually, some of the new *Socket* conversion functions `croak()` on bad arguments.) It is therefore essential to check return values from these functions.

Always begin your socket programs this way for optimal success, and don't forget to add `-T` taint checking flag to the `#!` line for servers:

```
#!/usr/bin/perl -Tw  
use strict;  
use sigtrap;  
use Socket;
```

BUGS

All these routines create system-specific portability problems. As noted elsewhere, Perl is at the mercy of your C libraries for much of its system behaviour. It's probably safest to assume broken SysV semantics for signals and to stick with simple TCP and UDP socket operations; e.g., don't try to pass open file descriptors over a local UDP datagram socket if you want your code to stand a chance of being portable.

As mentioned in the signals section, because few vendors provide C libraries that are safely re-entrant, the prudent programmer will do little else within a handler beyond setting a numeric variable that already exists; or, if locked into a slow (restarting) system call, using `die()` to raise an exception and `longjmp(3)` out. In fact, even these may in some cases cause a core dump. It's probably best to avoid signals except where they are absolutely inevitable. This will be addressed in a future release of Perl.

AUTHOR

Tom Christiansen, with occasional vestiges of Larry Wall's original version and suggestions from the Perl Porters.

SEE ALSO

There's a lot more to networking than this, but this should get you started.

For intrepid programmers, the indispensable textbook is *Unix Network Programming* by W. Richard Stevens (published by Addison-Wesley). Note that most books on networking address networking from the perspective of a C programmer; translation to Perl is left as an exercise for the reader.

The `IO::Socket(3)` manpage describes the object library, and the `Socket(3)` manpage describes the low-level interface to sockets. Besides the obvious functions in *perlfunc*, you should also check out the *modules* file at your nearest CPAN site. (See *perlmodlib* or best yet, the *Perl FAQ* for a description of what CPAN is and where to get it.)

Section 5 of the *modules* file is devoted to "Networking, Device Control (modems), and Interprocess Communication", and contains numerous unbundled modules numerous networking modules, Chat and Expect operations, CGI programming, DCE, FTP, IPC, NNTP, Proxy, Pty, RPC, SNMP, SMTP, Telnet, Threads, and ToolTalk—just to name a few.

NAME

perllexwarn – Perl Lexical Warnings

DESCRIPTION

The `use warnings` pragma is a replacement for both the command line flag `-w` and the equivalent Perl variable, `$^W`.

The pragma works just like the existing "strict" pragma. This means that the scope of the warning pragma is limited to the enclosing block. It also means that the pragma setting will not leak across files (via `use`, `require` or `do`). This allows authors to independently define the degree of warning checks that will be applied to their module.

By default, optional warnings are disabled, so any legacy code that doesn't attempt to control the warnings will work unchanged.

All warnings are enabled in a block by either of these:

```
use warnings ;
use warnings 'all' ;
```

Similarly all warnings are disabled in a block by either of these:

```
no warnings ;
no warnings 'all' ;
```

For example, consider the code below:

```
use warnings ;
my @a ;
{
    no warnings ;
    my $b = @a[0] ;
}
my $c = @a[0] ;
```

The code in the enclosing block has warnings enabled, but the inner block has them disabled. In this case that means the assignment to the scalar `$c` will trip the "Scalar value @a[0] better written as \$a[0]" warning, but the assignment to the scalar `$b` will not.

Default Warnings and Optional Warnings

Before the introduction of lexical warnings, Perl had two classes of warnings: mandatory and optional.

As its name suggests, if your code tripped a mandatory warning, you would get a warning whether you wanted it or not. For example, the code below would always produce an "isn't numeric" warning about the "2:".

```
my $a = "2:" + 3 ;
```

With the introduction of lexical warnings, mandatory warnings now become *default* warnings. The difference is that although the previously mandatory warnings are still enabled by default, they can then be subsequently enabled or disabled with the lexical warning pragma. For example, in the code below, an "isn't numeric" warning will only be reported for the `$a` variable.

```
my $a = "2:" + 3 ;
no warnings ;
my $b = "2:" + 3 ;
```

Note that neither the `-w` flag or the `$^W` can be used to disable/enable default warnings. They are still mandatory in this case.

What's wrong with `-w` and `$^W`

Although very useful, the big problem with using `-w` on the command line to enable warnings is that it is all or nothing. Take the typical scenario when you are writing a Perl program. Parts of the code you will write yourself, but it's very likely that you will make use of pre-written Perl modules. If you use the `-w` flag in this case, you end up enabling warnings in pieces of code that you haven't written.

Similarly, using `$^W` to either disable or enable blocks of code is fundamentally flawed. For a start, say you want to disable warnings in a block of code. You might expect this to be enough to do the trick:

```
{
    local ($^W) = 0 ;
    my $a += 2 ;
    my $b ; chop $b ;
}
```

When this code is run with the `-w` flag, a warning will be produced for the `$a` line — "Reversed += operator".

The problem is that Perl has both compile-time and run-time warnings. To disable compile-time warnings you need to rewrite the code like this:

```
{
    BEGIN { $^W = 0 }
    my $a += 2 ;
    my $b ; chop $b ;
}
```

The other big problem with `$^W` is the way you can inadvertently change the warning setting in unexpected places in your code. For example, when the code below is run (without the `-w` flag), the second call to `doit` will trip a "Use of uninitialized value" warning, whereas the first will not.

```
sub doit
{
    my $b ; chop $b ;
}

doit() ;

{
    local ($^W) = 1 ;
    doit()
}
```

This is a side-effect of `$^W` being dynamically scoped.

Lexical warnings get around these limitations by allowing finer control over where warnings can or can't be tripped.

Controlling Warnings from the Command Line

There are three Command Line flags that can be used to control when warnings are (or aren't) produced:

- w** This is the existing flag. If the lexical warnings pragma is **not** used in any of your code, or any of the modules that you use, this flag will enable warnings everywhere. See [Backward Compatibility](#) for details of how this flag interacts with lexical warnings.
- W** If the `-W` flag is used on the command line, it will enable all warnings throughout the program regardless of whether warnings were disabled locally using `no warnings` or `$^W = 0`. This includes all files that get included via `use`, `require` or `do`. Think of it as the Perl equivalent of the "lint" command.

-X Does the exact opposite to the **-W** flag, i.e. it disables all warnings.

Backward Compatibility

If you are used with working with a version of Perl prior to the introduction of lexically scoped warnings, or have code that uses both lexical warnings and `$^W`, this section will describe how they interact.

How Lexical Warnings interact with `-w/$^W`:

1. If none of the three command line flags (**-w**, **-W** or **-X**) that control warnings is used and neither `$^W` or the `warnings` pragma are used, then default warnings will be enabled and optional warnings disabled. This means that legacy code that doesn't attempt to control the warnings will work unchanged.
2. The **-w** flag just sets the global `$^W` variable as in 5.005 — this means that any legacy code that currently relies on manipulating `$^W` to control warning behavior will still work as is.
3. Apart from now being a boolean, the `$^W` variable operates in exactly the same horrible uncontrolled global way, except that it cannot disable/enable default warnings.
4. If a piece of code is under the control of the `warnings` pragma, both the `$^W` variable and the **-w** flag will be ignored for the scope of the lexical warning.
5. The only way to override a lexical warnings setting is with the **-W** or **-X** command line flags.

The combined effect of 3 & 4 is that it will allow code which uses the `warnings` pragma to control the warning behavior of `$^W`-type code (using a `local $^W=0`) if it really wants to, but not vice-versa.

Category Hierarchy

A hierarchy of "categories" have been defined to allow groups of warnings to be enabled/disabled in isolation.

The current hierarchy is:

```

all +-
    |
    +- closure
    |
    +- deprecated
    |
    +- exiting
    |
    +- glob
    |
    +- io -----+
    |             |
    |             +- closed
    |             |
    |             +- exec
    |             |
    |             +- layer
    |             |
    |             +- newline
    |             |
    |             +- pipe
    |             |
    |             +- unopened
    |
    +- misc
    |

```

```
+-- numeric
|
+-- once
|
+-- overflow
|
+-- pack
|
+-- portable
|
+-- recursion
|
+-- redefine
|
+-- regexp
|
+-- severe -----+
|                   |
|                   +- debugging
|                   |
|                   +- inplace
|                   |
|                   +- internal
|                   |
|                   +- malloc
|
+-- signal
|
+-- substr
|
+-- syntax -----+
|                   |
|                   +- ambiguous
|                   |
|                   +- bareword
|                   |
|                   +- digit
|                   |
|                   +- parenthesis
|                   |
|                   +- precedence
|                   |
|                   +- printf
|                   |
|                   +- prototype
|                   |
|                   +- qw
|                   |
|                   +- reserved
|                   |
|                   +- semicolon
|
+-- taint
|
```

```

+- threads
|
+- uninitialized
|
+- unpack
|
+- untie
|
+- utf8
|
+- void
|
+- y2k

```

Just like the "strict" pragma any of these categories can be combined

```

use warnings qw(void redefine) ;
no warnings qw(io syntax untie) ;

```

Also like the "strict" pragma, if there is more than one instance of the warnings pragma in a given scope the cumulative effect is additive.

```

use warnings qw(void) ; # only "void" warnings enabled
...
use warnings qw(io) ;   # only "void" & "io" warnings enabled
...
no warnings qw(void) ;  # only "io" warnings enabled

```

To determine which category a specific warning has been assigned to see [perldiag](#).

Note: In Perl 5.6.1, the lexical warnings category "deprecated" was a sub-category of the "syntax" category. It is now a top-level category in its own right.

Fatal Warnings

The presence of the word "FATAL" in the category list will escalate any warnings detected from the categories specified in the lexical scope into fatal errors. In the code below, the use of `time`, `length` and `join` can all produce a "Useless use of xxx in void context" warning.

```

use warnings ;

time ;

{
    use warnings FATAL => qw(void) ;
    length "abc" ;
}

join "", 1,2,3 ;

print "done\n" ;

```

When run it produces this output

```

Useless use of time in void context at fatal line 3.
Useless use of length in void context at fatal line 7.

```

The scope where `length` is used has escalated the void warnings category into a fatal error, so the program terminates immediately it encounters the warning.

To explicitly turn off a "FATAL" warning you just disable the warning it is associated with. So, for example, to disable the "void" warning in the example above, either of these will do the trick:

```
no warnings qw(void);
no warnings FATAL => qw(void);
```

If you want to downgrade a warning that has been escalated into a fatal error back to a normal warning, you can use the "NONFATAL" keyword. For example, the code below will promote all warnings into fatal errors, except for those in the "syntax" category.

```
use warnings FATAL => 'all', NONFATAL => 'syntax';
```

Reporting Warnings from a Module

The warnings pragma provides a number of functions that are useful for module authors. These are used when you want to report a module-specific warning to a calling module has enabled warnings via the warnings pragma.

Consider the module `MyMod:::Abc` below.

```
package MyMod:::Abc;
use warnings::register;
sub open {
    my $path = shift ;
    if ($path !~ m#^/#) {
        warnings::warn("changing relative path to /tmp/")
            if warnings::enabled();
        $path = "/tmp/$path" ;
    }
}
1 ;
```

The call to `warnings::register` will create a new warnings category called "MyMod:::abc", i.e. the new category name matches the current package name. The `open` function in the module will display a warning message if it gets given a relative path as a parameter. This warnings will only be displayed if the code that uses `MyMod:::Abc` has actually enabled them with the warnings pragma like below.

```
use MyMod:::Abc;
use warnings 'MyMod:::Abc';
...
abc::open("../fred.txt");
```

It is also possible to test whether the pre-defined warnings categories are set in the calling module with the `warnings::enabled` function. Consider this snippet of code:

```
package MyMod:::Abc;
sub open {
    warnings::warnif("deprecated",
                    "open is deprecated, use new instead") ;
    new(@_) ;
}
sub new
...
1 ;
```

The function `open` has been deprecated, so code has been included to display a warning message whenever the calling module has (at least) the "deprecated" warnings category enabled. Something like this, say.

```
use warnings 'deprecated';
use MyMod:::Abc;
...
```

```
MyMod::Abc::open($filename) ;
```

Either the `warnings::warn` or `warnings::warnif` function should be used to actually display the warnings message. This is because they can make use of the feature that allows warnings to be escalated into fatal errors. So in this case

```
use MyMod::Abc;
use warnings FATAL => 'MyMod::Abc';
...
MyMod::Abc::open('../fred.txt');
```

the `warnings::warnif` function will detect this and die after displaying the warning message.

The three warnings functions, `warnings::warn`, `warnings::warnif` and `warnings::enabled` can optionally take an object reference in place of a category name. In this case the functions will use the class name of the object as the warnings category.

Consider this example:

```
package Original ;

no warnings ;
use warnings::register ;

sub new
{
    my $class = shift ;
    bless [], $class ;
}

sub check
{
    my $self = shift ;
    my $value = shift ;

    if ($value % 2 && warnings::enabled($self))
    { warnings::warn($self, "Odd numbers are unsafe") }
}

sub doit
{
    my $self = shift ;
    my $value = shift ;
    $self->check($value) ;
    # ...
}

1 ;

package Derived ;

use warnings::register ;
use Original ;
our @ISA = qw( Original ) ;
sub new
{
    my $class = shift ;
    bless [], $class ;
}

1 ;
```

The code below makes use of both modules, but it only enables warnings from `Derived`.

```
use Original ;
use Derived ;
use warnings 'Derived';
my $a = new Original ;
$a->doit(1) ;
my $b = new Derived ;
$a->doit(1) ;
```

When this code is run only the `Derived` object, `$b`, will generate a warning.

```
Odd numbers are unsafe at main.pl line 7
```

Notice also that the warning is reported at the line where the object is first used.

TODO

`perl5db.pl`

The debugger saves and restores `C<$^W>` at runtime. I haven't checked whether the debugger will still work with the lexical warnings patch applied.

`diagnostics.pm`

I *think* I've got diagnostics to work with the lexical warnings patch, but there were design decisions made in diagnostics to work around the limitations of `C<$^W>`. Now that those limitations are gone, the module should be revisited.

document calling the `warnings::*` functions from XS

SEE ALSO

[warnings](#), [perldiag](#).

AUTHOR

Paul Marquess

NAME

perllocale – Perl locale handling (internationalization and localization)

DESCRIPTION

Perl supports language-specific notions of data such as "is this a letter", "what is the uppercase equivalent of this letter", and "which of these letters comes first". These are important issues, especially for languages other than English—but also for English: it would be naïve to imagine that A–Za–z defines all the "letters" needed to write in English. Perl is also aware that some character other than '.' may be preferred as a decimal point, and that output date representations may be language-specific. The process of making an application take account of its users' preferences in such matters is called **internationalization** (often abbreviated as **i18n**); telling such an application about a particular set of preferences is known as **localization** (**l10n**).

Perl can understand language-specific data via the standardized (ISO C, XPG4, POSIX 1.c) method called "the locale system". The locale system is controlled per application using one pragma, one function call, and several environment variables.

NOTE: This feature is new in Perl 5.004, and does not apply unless an application specifically requests it—see [Backward compatibility](#). The one exception is that `write()` now **always** uses the current locale – see ["NOTES"](#).

PREPARING TO USE LOCALES

If Perl applications are to understand and present your data correctly according a locale of your choice, **all** of the following must be true:

- **Your operating system must support the locale system.** If it does, you should find that the `setlocale()` function is a documented part of its C library.
- **Definitions for locales that you use must be installed.** You, or your system administrator, must make sure that this is the case. The available locales, the location in which they are kept, and the manner in which they are installed all vary from system to system. Some systems provide only a few, hard-wired locales and do not allow more to be added. Others allow you to add "canned" locales provided by the system supplier. Still others allow you or the system administrator to define and add arbitrary locales. (You may have to ask your supplier to provide canned locales that are not delivered with your operating system.) Read your system documentation for further illumination.
- **Perl must believe that the locale system is supported.** If it does, `perl -V:d_setlocale` will say that the value for `d_setlocale` is `define`.

If you want a Perl application to process and present your data according to a particular locale, the application code should include the `use locale` pragma (see [L<The use locale pragma](#)) where appropriate, and **at least one** of the following must be true:

- **The locale-determining environment variables (see ["ENVIRONMENT"](#)) must be correctly set up** at the time the application is started, either by yourself or by whoever set up your system account.
- **The application must set its own locale** using the method described in [The `setlocale` function](#).

USING LOCALES

The use locale pragma

By default, Perl ignores the current locale. The `use locale` pragma tells Perl to use the current locale for some operations:

- **The comparison operators** (`lt`, `le`, `cmp`, `ge`, and `gt`) and the POSIX string collation functions `strcoll()` and `strxfrm()` use `LC_COLLATE`. `sort()` is also affected if used without an explicit comparison function, because it uses `cmp` by default.

Note: `eq` and `ne` are unaffected by locale: they always perform a char-by-char comparison of their scalar operands. What's more, if `cmp` finds that its operands are equal according to the collation sequence specified by the current locale, it goes on to perform a char-by-char comparison, and only

returns (equal) if the operands are char-for-char identical. If you really want to know whether two strings—which eq and cmp may consider different—are equal as far as collation in the locale is concerned, see the discussion in [Category LC_COLLATE: Collation](#).

- **Regular expressions and case-modification functions** (uc(), lc(), ucfirst(), and lcfirst()) use LC_CTYPE
- **The formatting functions** (printf(), sprintf() and write()) use LC_NUMERIC
- **The POSIX date formatting function** (strftime()) uses LC_TIME.

LC_COLLATE, LC_CTYPE, and so on, are discussed further in [LOCALE CATEGORIES](#).

The default behavior is restored with the no locale pragma, or upon reaching the end of block enclosing use locale.

The string result of any operation that uses locale information is tainted, as it is possible for a locale to be untrustworthy. See ["SECURITY"](#).

The setlocale function

You can switch locales as often as you wish at run time with the POSIX::setlocale() function:

```
# This functionality not usable prior to Perl 5.004
require 5.004;

# Import locale-handling tool set from POSIX module.
# This example uses: setlocale -- the function call
#                      LC_CTYPE -- explained below
use POSIX qw(locale_h);

# query and save the old locale
$old_locale = setlocale(LC_CTYPE);

setlocale(LC_CTYPE, "fr_CA.ISO8859-1");
# LC_CTYPE now in locale "French, Canada, codeset ISO 8859-1"

setlocale(LC_CTYPE, "");
# LC_CTYPE now reset to default defined by LC_ALL/LC_CTYPE/LANG
# environment variables. See below for documentation.

# restore the old locale
setlocale(LC_CTYPE, $old_locale);
```

The first argument of setlocale() gives the **category**, the second the **locale**. The category tells in what aspect of data processing you want to apply locale-specific rules. Category names are discussed in [LOCALE CATEGORIES](#) and ["ENVIRONMENT"](#). The locale is the name of a collection of customization information corresponding to a particular combination of language, country or territory, and codeset. Read on for hints on the naming of locales: not all systems name locales as in the example.

If no second argument is provided and the category is something else than LC_ALL, the function returns a string naming the current locale for the category. You can use this value as the second argument in a subsequent call to setlocale().

If no second argument is provided and the category is LC_ALL, the result is implementation-dependent. It may be a string of concatenated locales names (separator also implementation-dependent) or a single locale name. Please consult your [setlocale\(3\)](#) for details.

If a second argument is given and it corresponds to a valid locale, the locale for the category is set to that value, and the function returns the now-current locale value. You can then use this in yet another call to setlocale(). (In some implementations, the return value may sometimes differ from the value you gave as the second argument—think of it as an alias for the value you gave.)

As the example shows, if the second argument is an empty string, the category's locale is returned to the

default specified by the corresponding environment variables. Generally, this results in a return to the default that was in force when Perl started up: changes to the environment made by the application after startup may or may not be noticed, depending on your system's C library.

If the second argument does not correspond to a valid locale, the locale for the category is not changed, and the function returns *undef*.

For further information about the categories, consult [setlocale\(3\)](#).

Finding locales

For locales available in your system, consult also [setlocale\(3\)](#) to see whether it leads to the list of available locales (search for the *SEE ALSO* section). If that fails, try the following command lines:

```
locale -a
nlsinfo
ls /usr/lib/nls/loc
ls /usr/lib/locale
ls /usr/lib/nls
ls /usr/share/locale
```

and see whether they list something resembling these

en_US.ISO8859-1	de_DE.ISO8859-1	ru_RU.ISO8859-5
en_US.iso88591	de_DE.iso88591	ru_RU.iso88595
en_US	de_DE	ru_RU
en	de	ru
english	german	russian
english.iso88591	german.iso88591	russian.iso88595
english.roman8		russian.koi8r

Sadly, even though the calling interface for `setlocale()` has been standardized, names of locales and the directories where the configuration resides have not been. The basic form of the name is *language_territory.codeset*, but the latter parts after *language* are not always present. The *language* and *country* are usually from the standards **ISO 3166** and **ISO 639**, the two-letter abbreviations for the countries and the languages of the world, respectively. The *codeset* part often mentions some **ISO 8859** character set, the Latin codesets. For example, ISO 8859-1 is the so-called "Western European codeset" that can be used to encode most Western European languages adequately. Again, there are several ways to write even the name of that one standard. Lamentably.

Two special locales are worth particular mention: "C" and "POSIX". Currently these are effectively the same locale: the difference is mainly that the first one is defined by the C standard, the second by the POSIX standard. They define the **default locale** in which every program starts in the absence of locale information in its environment. (The *default* default locale, if you will.) Its language is (American) English and its character codeset ASCII.

NOTE: Not all systems have the "POSIX" locale (not all systems are POSIX-conformant), so use "C" when you need explicitly to specify this default locale.

LOCALE PROBLEMS

You may encounter the following warning message at Perl startup:

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
    LC_ALL = "En_US",
    LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

This means that your locale settings had LC_ALL set to "En_US" and LANG exists but has no value. Perl tried to believe you but could not. Instead, Perl gave up and fell back to the "C" locale, the default locale that is supposed to work no matter what. This usually means your locale settings were wrong, they mention locales your system has never heard of, or the locale installation in your system has problems (for example, some system files are broken or missing). There are quick and temporary fixes to these problems, as well as more thorough and lasting fixes.

Temporarily fixing locale problems

The two quickest fixes are either to render Perl silent about any locale inconsistencies or to run Perl under the default locale "C".

Perl's moaning about locale problems can be silenced by setting the environment variable PERL_BADLANG to a zero value, for example "0". This method really just sweeps the problem under the carpet: you tell Perl to shut up even when Perl sees that something is wrong. Do not be surprised if later something locale-dependent misbehaves.

Perl can be run under the "C" locale by setting the environment variable LC_ALL to "C". This method is perhaps a bit more civilized than the PERL_BADLANG approach, but setting LC_ALL (or other locale variables) may affect other programs as well, not just Perl. In particular, external programs run from within Perl will see these changes. If you make the new settings permanent (read on), all programs you run see the changes. See [ENVIRONMENT](#) for the full list of relevant environment variables and [USING LOCALES](#) for their effects in Perl. Effects in other programs are easily deducible. For example, the variable LC_COLLATE may well affect your **sort** program (or whatever the program that arranges 'records' alphabetically in your system is called).

You can test out changing these variables temporarily, and if the new settings seem to help, put those settings into your shell startup files. Consult your local documentation for the exact details. For in Bourne-like shells (**sh**, **ksh**, **bash**, **zsh**):

```
LC_ALL=en_US.ISO8859-1
export LC_ALL
```

This assumes that we saw the locale "en_US.ISO8859-1" using the commands discussed above. We decided to try that instead of the above faulty locale "En_US"—and in Cshish shells (**csh**, **tcsh**)

```
setenv LC_ALL en_US.ISO8859-1
```

or if you have the "env" application you can do in any shell

```
env LC_ALL=en_US.ISO8859-1 perl ...
```

If you do not know what shell you have, consult your local helpdesk or the equivalent.

Permanently fixing locale problems

The slower but superior fixes are when you may be able to yourself fix the misconfiguration of your own environment variables. The mis(sing)configuration of the whole system's locales usually requires the help of your friendly system administrator.

First, see earlier in this document about [Finding locales](#). That tells how to find which locales are really supported—and more importantly, installed—on your system. In our example error message, environment variables affecting the locale are listed in the order of decreasing importance (and unset variables do not matter). Therefore, having LC_ALL set to "En_US" must have been the bad choice, as shown by the error message. First try fixing locale settings listed first.

Second, if using the listed commands you see something **exactly** (prefix matches do not count and case usually counts) like "En_US" without the quotes, then you should be okay because you are using a locale name that should be installed and available in your system. In this case, see [Permanently fixing your system's locale configuration](#).

Permanently fixing your system's locale configuration

This is when you see something like:

```
perl: warning: Please check that your locale settings:
    LC_ALL = "En_US",
    LANG = (unset)
are supported and installed on your system.
```

but then cannot see that "En_US" listed by the above-mentioned commands. You may see things like "en_US.ISO8859-1", but that isn't the same. In this case, try running under a locale that you can list and which somehow matches what you tried. The rules for matching locale names are a bit vague because standardization is weak in this area. See again the [Finding locales](#) about general rules.

Fixing system locale configuration

Contact a system administrator (preferably your own) and report the exact error message you get, and ask them to read this same documentation you are now reading. They should be able to check whether there is something wrong with the locale configuration of the system. The [Finding locales](#) section is unfortunately a bit vague about the exact commands and places because these things are not that standardized.

The localeconv function

The `POSIX::localeconv()` function allows you to get particulars of the locale-dependent numeric formatting information specified by the current `LC_NUMERIC` and `LC_MONETARY` locales. (If you just want the name of the current locale for a particular category, use `POSIX::setlocale()` with a single parameter—see [The setlocale function](#).)

```
use POSIX qw(locale_h);

# Get a reference to a hash of locale-dependent info
$locale_values = localeconv();

# Output sorted list of the values
for (sort keys %$locale_values) {
    printf "%-20s = %s\n", $_, $locale_values->{$_}
}
```

`localeconv()` takes no arguments, and returns **a reference to** a hash. The keys of this hash are variable names for formatting, such as `decimal_point` and `thousands_sep`. The values are the corresponding, er, values. See [localeconv](#) for a longer example listing the categories an implementation might be expected to provide; some provide more and others fewer. You don't need an explicit `use locale`, because `localeconv()` always observes the current locale.

Here's a simple-minded example program that rewrites its command-line parameters as integers correctly formatted in the current locale:

```
# See comments in previous example
require 5.004;
use POSIX qw(locale_h);

# Get some of locale's numeric formatting parameters
my ($thousands_sep, $grouping) =
    @{localeconv()}{'thousands_sep', 'grouping'};

# Apply defaults if values are missing
$thousands_sep = ',' unless $thousands_sep;

# grouping and mon_grouping are packed lists
# of small integers (characters) telling the
# grouping (thousand_seps and mon_thousand_seps
# being the group dividers) of numbers and
# monetary quantities. The integers' meanings:
```

```

# 255 means no more grouping, 0 means repeat
# the previous grouping, 1-254 means use that
# as the current grouping. Grouping goes from
# right to left (low to high digits). In the
# below we cheat slightly by never using anything
# else than the first grouping (whatever that is).
if ($grouping) {
    @grouping = unpack("C*", $grouping);
} else {
    @grouping = (3);
}

# Format command line params for current locale
for (@ARGV) {
    $_ = int;      # Chop non-integer part
    1 while
    s/(\d)(\d{$grouping[0]}($|thousands_sep))/ $1$thousands_sep$2/;
    print "$_";
}
print "\n";

```

I18N::Langinfo

Another interface for querying locale-dependent information is the `I18N::Langinfo::langinfo()` function, available at least in UNIX-like systems and VMS.

The following example will import the `langinfo()` function itself and three constants to be used as arguments to `langinfo()`: a constant for the abbreviated first day of the week (the numbering starts from Sunday = 1) and two more constants for the affirmative and negative answers for a yes/no question in the current locale.

```

use I18N::Langinfo qw(langinfo ABDAY_1 YESSTR NOSTR);
my ($abday_1, $yesstr, $nostr) = map { langinfo } qw(ABDAY_1 YESSTR NOSTR);
print "$abday_1? [$yesstr/$nostr] ";

```

In other words, in the "C" (or English) locale the above will probably print something like:

```
Sun? [yes/no]
```

See [I18N::Langinfo](#) for more information.

LOCALE CATEGORIES

The following subsections describe basic locale categories. Beyond these, some combination categories allow manipulation of more than one basic category at a time. See ["ENVIRONMENT"](#) for a discussion of these.

Category LC_COLLATE: Collation

In the scope of use `locale`, Perl looks to the `LC_COLLATE` environment variable to determine the application's notions on collation (ordering) of characters. For example, 'b' follows 'a' in Latin alphabets, but where do 'á' and 'â' belong? And while 'color' follows 'chocolate' in English, what about in Spanish?

The following collations all make sense and you may meet any of them if you "use locale".

```

A B C D E a b c d e
A a B b C c D d E e
a A b B c C d D e E
a b c d e A B C D E

```

Here is a code snippet to tell what "word" characters are in the current locale, in that locale's order:

```
use locale;
print +(sort grep /\w/, map { chr } 0..255), "\n";
```

Compare this with the characters that you see and their order if you state explicitly that the locale should be ignored:

```
no locale;
print +(sort grep /\w/, map { chr } 0..255), "\n";
```

This machine-native collation (which is what you get unless `use locale` has appeared earlier in the same block) must be used for sorting raw binary data, whereas the locale-dependent collation of the first example is useful for natural text.

As noted in [USING LOCALES](#), `cmp` compares according to the current collation locale when `use locale` is in effect, but falls back to a char-by-char comparison for strings that the locale says are equal. You can use `POSIX::strcoll()` if you don't want this fall-back:

```
use POSIX qw(strcoll);
$equal_in_locale =
    !strcoll("space and case ignored", "SpaceAndCaseIgnored");
```

`$equal_in_locale` will be true if the collation locale specifies a dictionary-like ordering that ignores space characters completely and which folds case.

If you have a single string that you want to check for "equality in locale" against several others, you might think you could gain a little efficiency by using `POSIX::strxfrm()` in conjunction with `eq`:

```
use POSIX qw(strxfrm);
$xfrm_string = strxfrm("Mixed-case string");
print "locale collation ignores spaces\n"
    if $xfrm_string eq strxfrm("Mixed-casestring");
print "locale collation ignores hyphens\n"
    if $xfrm_string eq strxfrm("Mixedcase string");
print "locale collation ignores case\n"
    if $xfrm_string eq strxfrm("mixed-case string");
```

`strxfrm()` takes a string and maps it into a transformed string for use in char-by-char comparisons against other transformed strings during collation. "Under the hood", locale-affected Perl comparison operators call `strxfrm()` for both operands, then do a char-by-char comparison of the transformed strings. By calling `strxfrm()` explicitly and using a non locale-affected comparison, the example attempts to save a couple of transformations. But in fact, it doesn't save anything: Perl magic (see [Magic Variables](#)) creates the transformed version of a string the first time it's needed in a comparison, then keeps this version around in case it's needed again. An example rewritten the easy way with `cmp` runs just about as fast. It also copes with null characters embedded in strings; if you call `strxfrm()` directly, it treats the first null it finds as a terminator. don't expect the transformed strings it produces to be portable across systems—or even from one revision of your operating system to the next. In short, don't call `strxfrm()` directly: let Perl do it for you.

Note: `use locale` isn't shown in some of these examples because it isn't needed: `strcoll()` and `strxfrm()` exist only to generate locale-dependent results, and so always obey the current `LC_COLLATE` locale.

Category LC_CTYPE: Character Types

In the scope of `use locale`, Perl obeys the `LC_CTYPE` locale setting. This controls the application's notion of which characters are alphabetic. This affects Perl's `\w` regular expression metanotation, which stands for alphanumeric characters—that is, alphabetic, numeric, and including other special characters such as the underscore or hyphen. (Consult [perlre](#) for

more information about regular expressions.) Thanks to LC_CTYPE, depending on your locale setting, characters like 'æ', 'ð', '`, and 'ø' may be understood as \w characters.

The LC_CTYPE locale also provides the map used in transliterating characters between lower and uppercase.

This affects the case-mapping functions—lc(), lcfirst(), uc(), and ucfirst(); case-mapping interpolation with \l, \L, \u, or \U in double-quoted strings and s/// substitutions; and case-independent regular expression pattern matching using the i modifier.

Finally, LC_CTYPE affects the POSIX character-class test functions—isalpha(), islower(), and so on. For example, if you move from the "C" locale to a 7-bit Scandinavian one, you may find—possibly to your surprise—that "|" moves from the ispunct() class to isalpha().

Note: A broken or malicious LC_CTYPE locale definition may result in clearly ineligible characters being considered to be alphanumeric by your application. For strict matching of (mundane) letters and digits—for example, in command strings—locale-aware applications should use \w inside a no locale block. See ["SECURITY"](#).

Category LC_NUMERIC: Numeric Formatting

In the scope of use locale, Perl obeys the LC_NUMERIC locale information, which controls an application's idea of how numbers should be formatted for human readability by the printf(), sprintf(), and write() functions. String-to-numeric conversion by the POSIX::strtod() function is also affected. In most implementations the only effect is to change the character used for the decimal point—perhaps from '.' to ','. These functions aren't aware of such niceties as thousands separation and so on. (See [The localeconv function](#) if you care about these things.)

Output produced by print() is also affected by the current locale: it depends on whether use locale or no locale is in effect, and corresponds to what you'd get from printf() in the "C" locale. The same is true for Perl's internal conversions between numeric and string formats:

```
use POSIX qw(strtod);
use locale;

$n = 5/2;    # Assign numeric 2.5 to $n

$a = " $n"; # Locale-dependent conversion to string

print "half five is $n\n";          # Locale-dependent output
printf "half five is %g\n", $n;     # Locale-dependent output
print "DECIMAL POINT IS COMMA\n"
    if $n == (strtod("2,5"))[0]; # Locale-dependent conversion
```

See also [I18N::Langinfo](#) and RADIXCHAR.

Category LC_MONETARY: Formatting of monetary amounts

The C standard defines the LC_MONETARY category, but no function that is affected by its contents. (Those with experience of standards committees will recognize that the working group decided to punt on the issue.)

Consequently, Perl takes no notice of it. If you really want to use LC_MONETARY, you can query its contents—see [The localeconv function](#)—and use the information that it returns in your application's own formatting of currency amounts. However, you may well find that the information, voluminous and complex though it may be, still does not quite meet your requirements: currency formatting is a hard nut to crack.

See also [I18N::Langinfo](#) and CRNCYSTR.

LC_TIME

Output produced by `POSIX::strftime()`, which builds a formatted human-readable date/time string, is affected by the current `LC_TIME` locale. Thus, in a French locale, the output produced by the `%B` format element (full month name) for the first month of the year would be "janvier". Here's how to get a list of long month names in the current locale:

```
use POSIX qw(strftime);
for (0..11) {
    $long_month_name[$_] =
        strftime("%B", 0, 0, 0, 1, $_, 96);
}
```

Note: `use locale` isn't needed in this example: as a function that exists only to generate locale-dependent results, `strftime()` always obeys the current `LC_TIME` locale.

See also [I18N::Langinfo](#) and `ABDAY_1..ABDAY_7`, `DAY_1..DAY_7`, `ABMON_1..ABMON_12`, and `ABMON_1..ABMON_12`.

Other categories

The remaining locale category, `LC_MESSAGES` (possibly supplemented by others in particular implementations) is not currently used by Perl—except possibly to affect the behavior of library functions called by extensions outside the standard Perl distribution and by the operating system and its utilities. Note especially that the string value of `$!` and the error messages given by external utilities may be changed by `LC_MESSAGES`. If you want to have portable error codes, use `%!`. See [Errno](#).

SECURITY

Although the main discussion of Perl security issues can be found in [perlsec](#), a discussion of Perl's locale handling would be incomplete if it did not draw your attention to locale-dependent security issues. Locales—particularly on systems that allow unprivileged users to build their own locales—are untrustworthy. A malicious (or just plain broken) locale can make a locale-aware application give unexpected results. Here are a few possibilities:

- Regular expression checks for safe file names or mail addresses using `\w` may be spoofed by an `LC_CTYPE` locale that claims that characters such as `>` and `|` are alphanumeric.
- String interpolation with case-mapping, as in, say, `$dest = "C:\U$name.$ext"`, may produce dangerous results if a bogus `LC_CTYPE` case-mapping table is in effect.
- A sneaky `LC_COLLATE` locale could result in the names of students with "D" grades appearing ahead of those with "A"s.
- An application that takes the trouble to use information in `LC_MONETARY` may format debits as if they were credits and vice versa if that locale has been subverted. Or it might make payments in US dollars instead of Hong Kong dollars.
- The date and day names in dates formatted by `strftime()` could be manipulated to advantage by a malicious user able to subvert the `LC_DATE` locale. ("Look—it says I wasn't in the building on Sunday.")

Such dangers are not peculiar to the locale system: any aspect of an application's environment which may be modified maliciously presents similar challenges. Similarly, they are not specific to Perl: any programming language that allows you to write programs that take account of their environment exposes you to these issues.

Perl cannot protect you from all possibilities shown in the examples—there is no substitute for your own vigilance—but, when `use locale` is in effect, Perl uses the tainting mechanism (see [perlsec](#)) to mark string results that become locale-dependent, and which may be untrustworthy in consequence. Here is a summary of the tainting behavior of operators and functions that may be affected by the locale:

- **Comparison operators** (`lt`, `le`, `ge`, `gt` and `cmp`):
Scalar true/false (or less/equal/greater) result is never tainted.
- **Case-mapping interpolation** (with `\l`, `\L`, `\u` or `\U`):
Result string containing interpolated material is tainted if `use locale` is in effect.
- **Matching operator** (`m//`):
Scalar true/false result never tainted.

Subpatterns, either delivered as a list-context result or as `$1` etc. are tainted if `use locale` is in effect, and the subpattern regular expression contains `\w` (to match an alphanumeric character), `\W` (non-alphanumeric character), `\s` (white-space character), or `\S` (non white-space character). The matched-pattern variable, `$&`, `$'` (pre-match), `$'` (post-match), and `$+` (last match) are also tainted if `use locale` is in effect and the regular expression contains `\w`, `\W`, `\s`, or `\S`.
- **Substitution operator** (`s//`):
Has the same behavior as the match operator. Also, the left operand of `=~` becomes tainted when `use locale` in effect if modified as a result of a substitution based on a regular expression match involving `\w`, `\W`, `\s`, or `\S`; or of case-mapping with `\l`, `\L`, `\u` or `\U`.
- **Output formatting functions** (`printf()` and `write()`):
Results are never tainted because otherwise even output from `print`, for example `print(1/7)`, should be tainted if `use locale` is in effect.
- **Case-mapping functions** (`lc()`, `lcfirst()`, `uc()`, `ucfirst()`):
Results are tainted if `use locale` is in effect.
- **POSIX locale-dependent functions** (`localeconv()`, `strcoll()`, `strftime()`, `strxfrm()`):
Results are never tainted.
- **POSIX character class tests** (`isalnum()`, `isalpha()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`):
True/false results are never tainted.

Three examples illustrate locale-dependent tainting. The first program, which ignores its locale, won't run: a value taken directly from the command line may not be used to name an output file when taint checks are enabled.

```
#!/usr/local/bin/perl -T
# Run with taint checking

# Command line sanity check omitted...
$tainted_output_file = shift;

open(F, ">$tainted_output_file")
    or warn "Open of $untainted_output_file failed: $!\n";
```

The program can be made to run by "laundering" the tainted value through a regular expression: the second example—which still ignores locale information—runs, creating the file named on its command line if it can.

```
#!/usr/local/bin/perl -T

$tainted_output_file = shift;
$tainted_output_file =~ m%[\w/]+%;
$untainted_output_file = $&;
```

```
open(F, ">$untainted_output_file")
    or warn "Open of $untainted_output_file failed: $!\n";
```

Compare this with a similar but locale-aware program:

```
#!/usr/local/bin/perl -T

$tainted_output_file = shift;
use locale;
$tainted_output_file =~ m%[\w/]+%;
$localized_output_file = $&;

open(F, ">$localized_output_file")
    or warn "Open of $localized_output_file failed: $!\n";
```

This third program fails to run because `$&` is tainted: it is the result of a match involving `\w` while `use locale` is in effect.

ENVIRONMENT

PERL_BADLANG

A string that can suppress Perl's warning about failed locale settings at startup. Failure can occur if the locale support in the operating system is lacking (broken) in some way—or if you mistyped the name of a locale when you set up your environment. If this environment variable is absent, or has a value that does not evaluate to integer zero—that is, "0" or ""—Perl will complain about locale setting failures.

NOTE: `PERL_BADLANG` only gives you a way to hide the warning message. The message tells about some problem in your system's locale support, and you should investigate what the problem is.

The following environment variables are not specific to Perl: They are part of the standardized (ISO C, XPG4, POSIX 1.c) `setlocale()` method for controlling an application's opinion on data.

LC_ALL `LC_ALL` is the "override-all" locale environment variable. If set, it overrides all the rest of the locale environment variables.

LANGUAGE **NOTE:** `LANGUAGE` is a GNU extension, it affects you only if you are using the GNU libc. This is the case if you are using e.g. Linux. If you are using "commercial" UNIXes you are most probably *not* using GNU libc and you can ignore `LANGUAGE`.

However, in the case you are using `LANGUAGE`: it affects the language of informational, warning, and error messages output by commands (in other words, it's like `LC_MESSAGES`) but it has higher priority than [LC_ALL](#). Moreover, it's not a single value but instead a "path" (":"-separated list) of *languages* (not locales). See the GNU `gettext` library documentation for more information.

LC_CTYPE In the absence of `LC_ALL`, `LC_CTYPE` chooses the character type locale. In the absence of both `LC_ALL` and `LC_CTYPE`, `LANG` chooses the character type locale.

LC_COLLATE In the absence of `LC_ALL`, `LC_COLLATE` chooses the collation (sorting) locale. In the absence of both `LC_ALL` and `LC_COLLATE`, `LANG` chooses the collation locale.

LC_MONETARY

In the absence of `LC_ALL`, `LC_MONETARY` chooses the monetary formatting locale. In the absence of both `LC_ALL` and `LC_MONETARY`, `LANG` chooses the monetary formatting locale.

LC_NUMERIC In the absence of `LC_ALL`, `LC_NUMERIC` chooses the numeric format locale. In the absence of both `LC_ALL` and `LC_NUMERIC`, `LANG` chooses the numeric format.

LC_TIME	In the absence of LC_ALL, LC_TIME chooses the date and time formatting locale. In the absence of both LC_ALL and LC_TIME, LANG chooses the date and time formatting locale.
LANG	LANG is the "catch-all" locale environment variable. If it is set, it is used as the last resort after the overall LC_ALL and the category-specific LC_ . . .

NOTES

Backward compatibility

Versions of Perl prior to 5.004 **mostly** ignored locale information, generally behaving as if something similar to the "C" locale were always in force, even if the program environment suggested otherwise (see [The *setlocale* function](#)). By default, Perl still behaves this way for backward compatibility. If you want a Perl application to pay attention to locale information, you **must** use the `use locale` pragma (see [The *use locale* pragma](#)) to instruct it to do so.

Versions of Perl from 5.002 to 5.003 did use the LC_CTYPE information if available; that is, `\w` did understand what were the letters according to the locale environment variables. The problem was that the user had no control over the feature: if the C library supported locales, Perl used them.

I18N:Collate obsolete

In versions of Perl prior to 5.004, per-locale collation was possible using the `I18N::Collate` library module. This module is now mildly obsolete and should be avoided in new applications. The LC_COLLATE functionality is now integrated into the Perl core language: One can use locale-specific scalar data completely normally with `use locale`, so there is no longer any need to juggle with the scalar references of `I18N::Collate`.

Sort speed and memory use impacts

Comparing and sorting by locale is usually slower than the default sorting; slow-downs of two to four times have been observed. It will also consume more memory: once a Perl scalar variable has participated in any string comparison or sorting operation obeying the locale collation rules, it will take 3–15 times more memory than before. (The exact multiplier depends on the string's contents, the operating system and the locale.) These downsides are dictated more by the operating system's implementation of the locale system than by Perl.

`write()` and LC_NUMERIC

Formats are the only part of Perl that unconditionally use information from a program's locale; if a program's environment specifies an LC_NUMERIC locale, it is always used to specify the decimal point character in formatted output. Formatted output cannot be controlled by `use locale` because the pragma is tied to the block structure of the program, and, for historical reasons, formats exist outside that block structure.

Freely available locale definitions

There is a large collection of locale definitions at <ftp://dkuug.dk/i18n/WG15-collection>. You should be aware that it is unsupported, and is not claimed to be fit for any purpose. If your system allows installation of arbitrary locales, you may find the definitions useful as they are, or as a basis for the development of your own locales.

I18n and I10n

"Internationalization" is often abbreviated as **i18n** because its first and last letters are separated by eighteen others. (You may guess why the internalin ... internaliti ... i18n tends to get abbreviated.) In the same way, "localization" is often abbreviated to **l10n**.

An imperfect standard

Internationalization, as defined in the C and POSIX standards, can be criticized as incomplete, ungainly, and having too large a granularity. (Locales apply to a whole process, when it would arguably be more useful to have them apply to a single thread, window group, or whatever.) They also have a tendency, like standards groups, to divide the world into nations, when we all know that the world can equally well be divided into

bankers, bikers, gamers, and so on. But, for now, it's the only standard we've got. This may be construed as a bug.

Unicode and UTF-8

The support of Unicode is new starting from Perl version 5.6, and more fully implemented in the version 5.8. See [perluniintro](#) and [perlunicode](#) for more details.

Usually locale settings and Unicode do not affect each other, but there are exceptions, see [Locales](#) for examples.

BUGS

Broken systems

In certain systems, the operating system's locale support is broken and cannot be fixed or used by Perl. Such deficiencies can and will result in mysterious hangs and/or Perl core dumps when the `use locale` is in effect. When confronted with such a system, please report in excruciating detail to [<perlbug@perl.org>](mailto:perlbug@perl.org), and complain to your vendor: bug fixes may exist for these problems in your operating system. Sometimes such bug fixes are called an operating system upgrade.

SEE ALSO

[I18N::Langinfo](#), [perluniintro](#), [perlunicode](#), [open](#), [isalnum](#), [isalpha](#), [isdigit](#), [isgraph](#), [islower](#), [isprint](#), [ispunct](#), [isspace](#), [isupper](#), [isxdigit](#), [localeconv](#), [setlocale](#), [strcoll](#), [strftime](#), [strtod](#), [strxfrm](#).

HISTORY

Jarkko Hietaniemi's original *perl18n.pod* heavily hacked by Dominic Dunlop, assisted by the perl5-porters. Prose worked over a bit by Tom Christiansen.

Last update: Thu Jun 11 08:44:13 MDT 1998

NAME

perllo1 – Manipulating Arrays of Arrays in Perl

DESCRIPTION**Declaration and Access of Arrays of Arrays**

The simplest thing to build is an array of arrays (sometimes imprecisely called a list of lists). It's reasonably easy to understand, and almost everything that applies here will also be applicable later on with the fancier data structures.

An array of an array is just a regular old array @AoA that you can get at with two subscripts, like \$AoA[3][2]. Here's a declaration of the array:

```
# assign to our array, an array of array references
@AoA = (
    [ "fred", "barney" ],
    [ "george", "jane", "elroy" ],
    [ "homer", "marge", "bart" ],
);

print $AoA[2][2];
bart
```

Now you should be very careful that the outer bracket type is a round one, that is, a parenthesis. That's because you're assigning to an @array, so you need parentheses. If you wanted there *not* to be an @AoA, but rather just a reference to it, you could do something more like this:

```
# assign a reference to array of array references
$ref_to_AoA = [
    [ "fred", "barney", "pebbles", "bambam", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $ref_to_AoA->[2][2];
```

Notice that the outer bracket type has changed, and so our access syntax has also changed. That's because unlike C, in perl you can't freely interchange arrays and references thereto. \$ref_to_AoA is a reference to an array, whereas @AoA is an array proper. Likewise, \$AoA[2] is not an array, but an array ref. So how come you can write these:

```
$AoA[2][2]
$ref_to_AoA->[2][2]
```

instead of having to write these:

```
$AoA[2]->[2]
$ref_to_AoA->[2]->[2]
```

Well, that's because the rule is that on adjacent brackets only (whether square or curly), you are free to omit the pointer dereferencing arrow. But you cannot do so for the very first one if it's a scalar containing a reference, which means that \$ref_to_AoA always needs it.

Growing Your Own

That's all well and good for declaration of a fixed data structure, but what if you wanted to add new elements on the fly, or build it up entirely from scratch?

First, let's look at reading it in from a file. This is something like adding a row at a time. We'll assume that there's a flat file in which each line is a row and each word an element. If you're trying to develop an @AoA array containing all these, here's the right way to do that:

```
while (<>) {
    @tmp = split;
    push @AoA, [ @tmp ];
}
```

You might also have loaded that from a function:

```
for $i ( 1 .. 10 ) {
    $AoA[$i] = [ somefunc($i) ];
}
```

Or you might have had a temporary variable sitting around with the array in it.

```
for $i ( 1 .. 10 ) {
    @tmp = somefunc($i);
    $AoA[$i] = [ @tmp ];
}
```

It's very important that you make sure to use the `[]` array reference constructor. That's because this will be very wrong:

```
$AoA[$i] = @tmp;
```

You see, assigning a named array like that to a scalar just counts the number of elements in `@tmp`, which probably isn't what you want.

If you are running under `use strict`, you'll have to add some declarations to make it happy:

```
use strict;
my(@AoA, @tmp);
while (<>) {
    @tmp = split;
    push @AoA, [ @tmp ];
}
```

Of course, you don't need the temporary array to have a name at all:

```
while (<>) {
    push @AoA, [ split ];
}
```

You also don't have to use `push()`. You could just make a direct assignment if you knew where you wanted to put it:

```
my (@AoA, $i, $line);
for $i ( 0 .. 10 ) {
    $line = <>;
    $AoA[$i] = [ split ' ', $line ];
}
```

or even just

```
my (@AoA, $i);
for $i ( 0 .. 10 ) {
    $AoA[$i] = [ split ' ', <> ];
}
```

You should in general be leery of using functions that could potentially return lists in scalar context without explicitly stating such. This would be clearer to the casual reader:

```
my (@AoA, $i);
for $i ( 0 .. 10 ) {
    $AoA[$i] = [ split ' ', scalar(<>) ];
}
```

```
}
```

If you wanted to have a `$ref_to_AoA` variable as a reference to an array, you'd have to do something like this:

```
while (<>) {
    push @$ref_to_AoA, [ split ];
}
```

Now you can add new rows. What about adding new columns? If you're dealing with just matrices, it's often easiest to use simple assignment:

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        $AoA[$x][$y] = func($x, $y);
    }
}

for $x ( 3, 7, 9 ) {
    $AoA[$x][20] += func2($x);
}
```

It doesn't matter whether those elements are already there or not: it'll gladly create them for you, setting intervening elements to `undef` as need be.

If you wanted just to append to a row, you'd have to do something a bit funnier looking:

```
# add new columns to an existing row
push @ { $AoA[0] }, "wilma", "betty";
```

Notice that I *couldn't* say just:

```
push $AoA[0], "wilma", "betty"; # WRONG!
```

In fact, that wouldn't even compile. How come? Because the argument to `push()` must be a real array, not just a reference to such.

Access and Printing

Now it's time to print your data structure out. How are you going to do that? Well, if you want only one of the elements, it's trivial:

```
print $AoA[0][0];
```

If you want to print the whole thing, though, you can't say

```
print @AoA; # WRONG
```

because you'll get just references listed, and perl will never automatically dereference things for you. Instead, you have to roll yourself a loop or two. This prints the whole structure, using the shell-style `for()` construct to loop across the outer set of subscripts.

```
for $aref ( @AoA ) {
    print "\t [ @$aref ],\n";
}
```

If you wanted to keep track of subscripts, you might do this:

```
for $i ( 0 .. $#AoA ) {
    print "\t elt $i is [ @{$AoA[$i]} ],\n";
}
```

or maybe even this. Notice the inner loop.

```
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. ${#AoA[$i]} ) {
```

```

        print "elt $i $j is $AoA[$i][$j]\n";
    }
}

```

As you can see, it's getting a bit complicated. That's why sometimes is easier to take a temporary on your way through:

```

for $i ( 0 .. $#AoA ) {
    $aref = $AoA[$i];
    for $j ( 0 .. ${$aref} ) {
        print "elt $i $j is $AoA[$i][$j]\n";
    }
}

```

Hmm... that's still a bit ugly. How about this:

```

for $i ( 0 .. $#AoA ) {
    $aref = $AoA[$i];
    $n = @$aref - 1;
    for $j ( 0 .. $n ) {
        print "elt $i $j is $AoA[$i][$j]\n";
    }
}

```

Slices

If you want to get at a slice (part of a row) in a multidimensional array, you're going to have to do some fancy subscripting. That's because while we have a nice synonym for single elements via the pointer arrow for dereferencing, no such convenience exists for slices. (Remember, of course, that you can always write a loop to do a slice operation.)

Here's how to do one operation using a loop. We'll assume an @AoA variable as before.

```

@part = ();
$x = 4;
for ($y = 7; $y < 13; $y++) {
    push @part, $AoA[$x][$y];
}

```

That same loop could be replaced with a slice operation:

```

@part = @{ $AoA[4] } [ 7..12 ];

```

but as you might well imagine, this is pretty rough on the reader.

Ah, but what if you wanted a *two-dimensional slice*, such as having \$x run from 4..8 and \$y run from 7 to 12? Hmm... here's the simple way:

```

@newAoA = ();
for ($startx = $x = 4; $x <= 8; $x++) {
    for ($starty = $y = 7; $y <= 12; $y++) {
        $newAoA[$x - $startx][$y - $starty] = $AoA[$x][$y];
    }
}

```

We can reduce some of the looping through slices

```

for ($x = 4; $x <= 8; $x++) {
    push @newAoA, [ @{ $AoA[$x] } [ 7..12 ] ];
}

```

If you were into Schwartzian Transforms, you would probably have selected map for that


```
@newAoA = map { [ @{ $AoA[$_] } [ 7..12 ] ] } 4 .. 8;
```

Although if your manager accused of seeking job security (or rapid insecurity) through inscrutable code, it would be hard to argue. :-) If I were you, I'd put that in a function:

```
@newAoA = splice_2D( \@AoA, 4 => 8, 7 => 12 );
sub splice_2D {
    my $lrr = shift;          # ref to array of array refs!
    my ($x_lo, $x_hi,
        $y_lo, $y_hi) = @_;

    return map {
        [ @{ $lrr->[$_] } [ $y_lo .. $y_hi ] ]
    } $x_lo .. $x_hi;
}
```

SEE ALSO

perldata(1), perlref(1), perldsc(1)

AUTHOR

Tom Christiansen <*tchrist@perl.com*>

Last update: Thu Jun 4 16:16:23 MDT 1998

NAME

perlmod – Perl modules (packages and symbol tables)

DESCRIPTION

Packages

Perl provides a mechanism for alternative namespaces to protect packages from stomping on each other's variables. In fact, there's really no such thing as a global variable in Perl. The package statement declares the compilation unit as being in the given namespace. The scope of the package declaration is from the declaration itself through the end of the enclosing block, `eval`, or file, whichever comes first (the same scope as the `my()` and `local()` operators). Unqualified dynamic identifiers will be in this namespace, except for those few identifiers that if unqualified, default to the main package instead of the current one as described below. A package statement affects only dynamic variables—including those you've used `local()` on—but *not* lexical variables created with `my()`. Typically it would be the first declaration in a file included by the `do`, `require`, or `use` operators. You can switch into a package in more than one place; it merely influences which symbol table is used by the compiler for the rest of that block. You can refer to variables and filehandles in other packages by prefixing the identifier with the package name and a double colon: `$Package::Variable`. If the package name is null, the main package is assumed. That is, `$::sail` is equivalent to `$main::sail`.

The old package delimiter was a single quote, but double colon is now the preferred delimiter, in part because it's more readable to humans, and in part because it's more readable to **emacs** macros. It also makes C++ programmers feel like they know what's going on—as opposed to using the single quote as separator, which was there to make Ada programmers feel like they knew what's going on. Because the old-fashioned syntax is still supported for backwards compatibility, if you try to use a string like "This is \$owner's house", you'll be accessing `$owner::s`; that is, the `$s` variable in package `owner`, which is probably not what you meant. Use braces to disambiguate, as in "This is \${owner}'s house".

Packages may themselves contain package separators, as in `$OUTER::INNER::var`. This implies nothing about the order of name lookups, however. There are no relative packages: all symbols are either local to the current package, or must be fully qualified from the outer package name down. For instance, there is nowhere within package `OUTER` that `$INNER::var` refers to `$OUTER::INNER::var`. It would treat package `INNER` as a totally separate global package.

Only identifiers starting with letters (or underscore) are stored in a package's symbol table. All other symbols are kept in package `main`, including all punctuation variables, like `$_`. In addition, when unqualified, the identifiers `STDIN`, `STDOUT`, `STDERR`, `ARGV`, `ARGVOUT`, `ENV`, `INC`, and `SIG` are forced to be in package `main`, even when used for other purposes than their built-in one. If you have a package called `m`, `s`, or `y`, then you can't use the qualified form of an identifier because it would be instead interpreted as a pattern match, a substitution, or a transliteration.

Variables beginning with underscore used to be forced into package `main`, but we decided it was more useful for package writers to be able to use leading underscore to indicate private variables and method names. `$_` is still global though. See also [Technical Note on the Syntax of Variable Names in perlvar](#).

eval'd strings are compiled in the package in which the `eval()` was compiled. (Assignments to `$SIG{}`, however, assume the signal handler specified is in the main package. Qualify the signal handler name if you wish to have a signal handler in a package.) For an example, examine *perl_{db}.pl* in the Perl library. It initially switches to the `DB` package so that the debugger doesn't interfere with variables in the program you are trying to debug. At various points, however, it temporarily switches back to the main package to evaluate various expressions in the context of the main package (or wherever you came from). See *perl_{debug}*.

The special symbol `__PACKAGE__` contains the current package, but cannot (easily) be used to construct variables.

See *perl_{sub}* for other scoping issues related to `my()` and `local()`, and *perl_{ref}* regarding closures.

Symbol Tables

The symbol table for a package happens to be stored in the hash of that name with two colons appended. The main symbol table's name is thus `%main::`, or `%::` for short. Likewise the symbol table for the nested package mentioned earlier is named `%OUTER::INNER::`.

The value in each entry of the hash is what you are referring to when you use the `*name` typeglob notation. In fact, the following have the same effect, though the first is more efficient because it does the symbol table lookups at compile time:

```
local *main::foo    = *main::bar;
local $main::{foo} = $main::{bar};
```

(Be sure to note the **vast** difference between the second line above and `local $main::foo = $main::bar`. The former is accessing the hash `%main::`, which is the symbol table of package `main`. The latter is simply assigning scalar `$bar` in package `main` to scalar `$foo` of the same package.)

You can use this to print out all the variables in a package, for instance. The standard but antiquated *dumpvar.pl* library and the CPAN module `Devel::Symdump` make use of this.

Assignment to a typeglob performs an aliasing operation, i.e.,

```
*dick = *richard;
```

causes variables, subroutines, formats, and file and directory handles accessible via the identifier `richard` also to be accessible via the identifier `dick`. If you want to alias only a particular variable or subroutine, assign a reference instead:

```
*dick = \$richard;
```

Which makes `$richard` and `$dick` the same variable, but leaves `@richard` and `@dick` as separate arrays. Tricky, eh?

There is one subtle difference between the following statements:

```
*foo = *bar;
*foo = \$bar;
```

`*foo = *bar` makes the typeglobs themselves synonymous while `*foo = \$bar` makes the SCALAR portions of two distinct typeglobs refer to the same scalar value. This means that the following code:

```
$bar = 1;
*foo = \$bar;          # Make $foo an alias for $bar
{
    local $bar = 2; # Restrict changes to block
    print $foo;     # Prints '1'!
}
```

Would print `'1'`, because `$foo` holds a reference to the *original* `$bar` — the one that was stuffed away by `local()` and which will be restored when the block ends. Because variables are accessed through the typeglob, you can use `*foo = *bar` to create an alias which can be localized. (But be aware that this means you can't have a separate `@foo` and `@bar`, etc.)

What makes all of this important is that the `Exporter` module uses glob aliasing as the import/export mechanism. Whether or not you can properly localize a variable that has been exported from a module depends on how it was exported:

```
@EXPORT = qw($FOO); # Usual form, can't be localized
@EXPORT = qw(*FOO); # Can be localized
```

You can work around the first case by using the fully qualified name (`$Package::FOO`) where you need a local value, or by overriding it by saying `*FOO = *Package::FOO` in your script.

The `*x = \%y` mechanism may be used to pass and return cheap references into or from subroutines if you don't want to copy the whole thing. It only works when assigning to dynamic variables, not lexicals.

```
%some_hash = (); # can't be my()
*some_hash = fn( \%another_hash );
sub fn {
    local *hashsym = shift;
    # now use %hashsym normally, and you
    # will affect the caller's %another_hash
    my %nhash = (); # do what you want
    return \%nhash;
}
```

On return, the reference will overwrite the hash slot in the symbol table specified by the `*some_hash` typeglob. This is a somewhat tricky way of passing around references cheaply when you don't want to have to remember to dereference variables explicitly.

Another use of symbol tables is for making "constant" scalars.

```
*PI = \%3.14159265358979;
```

Now you cannot alter `$PI`, which is probably a good thing all in all. This isn't the same as a constant subroutine, which is subject to optimization at compile-time. A constant subroutine is one prototyped to take no arguments and to return a constant expression. See [perlsyn](#) for details on these. The use constant pragma is a convenient shorthand for these.

You can say `*foo{PACKAGE}` and `*foo{NAME}` to find out what name and package the `*foo` symbol table entry comes from. This may be useful in a subroutine that gets passed typeglobs as arguments:

```
sub identify_typeglob {
    my $glob = shift;
    print 'You gave me ', *{$glob}{PACKAGE}, ' :: ', *{$glob}{NAME}, "\n";
}
identify_typeglob *foo;
identify_typeglob *bar::baz;
```

This prints

```
You gave me main::foo
You gave me bar::baz
```

The `*foo{THING}` notation can also be used to obtain references to the individual elements of `*foo`. See [perlref](#).

Subroutine definitions (and declarations, for that matter) need not necessarily be situated in the package whose symbol table they occupy. You can define a subroutine outside its package by explicitly qualifying the name of the subroutine:

```
package main;
sub Some_package::foo { ... } # &foo defined in Some_package
```

This is just a shorthand for a typeglob assignment at compile time:

```
BEGIN { *Some_package::foo = sub { ... } }
```

and is *not* the same as writing:

```
{
    package Some_package;
    sub foo { ... }
}
```

In the first two versions, the body of the subroutine is lexically in the main package, *not* in `Some_package`. So something like this:

```
package main;

$Some_package::name = "fred";
$main::name = "barney";

sub Some_package::foo {
    print "in ", __PACKAGE__, ": \"$name is '$name'\\n";
}

Some_package::foo();
```

prints:

```
in main: $name is 'barney'
```

rather than:

```
in Some_package: $name is 'fred'
```

This also has implications for the use of the `SUPER::` qualifier (see [perlobj](#)).

Package Constructors and Destructors

Four special subroutines act as package constructors and destructors. These are the `BEGIN`, `CHECK`, `INIT`, and `END` routines. The `sub` is optional for these routines.

A `BEGIN` subroutine is executed as soon as possible, that is, the moment it is completely defined, even before the rest of the containing file is parsed. You may have multiple `BEGIN` blocks within a file—they will execute in order of definition. Because a `BEGIN` block executes immediately, it can pull in definitions of subroutines and such from other files in time to be visible to the rest of the file. Once a `BEGIN` has run, it is immediately undefined and any code it used is returned to Perl's memory pool. This means you can't ever explicitly call a `BEGIN`.

An `END` subroutine is executed as late as possible, that is, after perl has finished running the program and just before the interpreter is being exited, even if it is exiting as a result of a `die()` function. (But not if it's polymorphing into another program via `exec`, or being blown out of the water by a signal—you have to trap that yourself (if you can).) You may have multiple `END` blocks within a file—they will execute in reverse order of definition; that is: last in, first out (LIFO). `END` blocks are not executed when you run perl with the `-c` switch, or if compilation fails.

Inside an `END` subroutine, `$?` contains the value that the program is going to pass to `exit()`. You can modify `$?` to change the exit value of the program. Beware of changing `$?` by accident (e.g. by running something via `system`).

Similar to `BEGIN` blocks, `INIT` blocks are run just before the Perl runtime begins execution, in "first in, first out" (FIFO) order. For example, the code generators documented in [perlcc](#) make use of `INIT` blocks to initialize and resolve pointers to XSUBs.

Similar to `END` blocks, `CHECK` blocks are run just after the Perl compile phase ends and before the run time begins, in LIFO order. `CHECK` blocks are again useful in the Perl compiler suite to save the compiled state of the program.

When you use the `-n` and `-p` switches to Perl, `BEGIN` and `END` work just as they do in **awk**, as a degenerate case. Both `BEGIN` and `CHECK` blocks are run when you use the `-c` switch for a compile-only syntax check, although your main code is not.

Perl Classes

There is no special class syntax in Perl, but a package may act as a class if it provides subroutines to act as methods. Such a package may also derive some of its methods from another class (package) by listing the other package name(s) in its global `@ISA` array (which must be a package global, not a lexical).

For more on this, see [perltoot](#) and [perlobj](#).

Perl Modules

A module is just a set of related functions in a library file, i.e., a Perl package with the same name as the file. It is specifically designed to be reusable by other modules or programs. It may do this by providing a mechanism for exporting some of its symbols into the symbol table of any package using it. Or it may function as a class definition and make its semantics available implicitly through method calls on the class and its objects, without explicitly exporting anything. Or it can do a little of both.

For example, to start a traditional, non-OO module called `Some::Module`, create a file called *Some/Module.pm* and start with this template:

```
package Some::Module; # assumes Some/Module.pm

use strict;
use warnings;

BEGIN {
    use Exporter ();
    our ($VERSION, @ISA, @EXPORT, @EXPORT_OK, %EXPORT_TAGS);

    # set the version for version checking
    $VERSION = 1.00;
    # if using RCS/CVS, this may be preferred
    $VERSION = do { my @r = (q$Revision: 2.21 $ =~ /\d+/g); sprintf "%d"."%02d"

    @ISA = qw(Exporter);
    @EXPORT = qw(&func1 &func2 &func4);
    %EXPORT_TAGS = ( ); # eg: TAG => [ qw!name1 name2! ],

    # your exported package globals go here,
    # as well as any optionally exported functions
    @EXPORT_OK = qw($Var1 %Hashit &func3);
}
our @EXPORT_OK;

# exported package globals go here
our $Var1;
our %Hashit;

# non-exported package globals go here
our @more;
our $stuff;

# initialize package globals, first exported ones
$Var1 = '';
%Hashit = ();

# then the others (which are still accessible as $Some::Module::stuff)
$stuff = '';
@more = ();

# all file-scoped lexicals must be created before
# the functions below that use them.

# file-private lexicals go here
my $priv_var = '';
my %secret_hash = ();

# here's a file-private function as a closure,
# callable as &$priv_func; it cannot be prototyped.
my $priv_func = sub {
```

```

        # stuff goes here.
    };

    # make all your functions, whether exported or not;
    # remember to put something interesting in the {} stubs
    sub func1      {}      # no prototype
    sub func2()    {}      # proto'd void
    sub func3($$)  {}      # proto'd to 2 scalars

    # this one isn't exported, but could be called!
    sub func4(\%)  {}      # proto'd to 1 hash ref

    END { }        # module clean-up code here (global destructor)

    ## YOUR CODE GOES HERE

    1; # don't forget to return a true value from the file

```

Then go on to declare and use your variables in functions without any qualifications. See [Exporter](#) and the [perlmodlib](#) for details on mechanics and style issues in module creation.

Perl modules are included into your program by saying

```
use Module;
```

or

```
use Module LIST;
```

This is exactly equivalent to

```
BEGIN { require Module; import Module; }
```

or

```
BEGIN { require Module; import Module LIST; }
```

As a special case

```
use Module ();
```

is exactly equivalent to

```
BEGIN { require Module; }
```

All Perl module files have the extension **.pm**. The `use` operator assumes this so you don't have to spell out "**Module.pm**" in quotes. This also helps to differentiate new modules from old **.pl** and **.ph** files. Module names are also capitalized unless they're functioning as pragmas; pragmas are in effect compiler directives, and are sometimes called "pragmatic modules" (or even "pragmata" if you're a classicist).

The two statements:

```
require SomeModule;
require "SomeModule.pm";
```

differ from each other in two ways. In the first case, any double colons in the module name, such as `Some::Module`, are translated into your system's directory separator, usually `/`. The second case does not, and would have to be specified literally. The other difference is that seeing the first `require` clues in the compiler that uses of indirect object notation involving `"SomeModule"`, as in `$ob = purge SomeModule`, are method calls, not function calls. (Yes, this really can make a difference.)

Because the `use` statement implies a `BEGIN` block, the importing of semantics happens as soon as the `use` statement is compiled, before the rest of the file is compiled. This is how it is able to function as a pragma mechanism, and also how modules are able to declare subroutines that are then visible as list or unary operators for the rest of the current file. This will not work if you use `require` instead of `use`. With `require` you can get into this problem:

```

require Cwd;          # make Cwd:: accessible
$here = Cwd::getcwd();

use Cwd;              # import names from Cwd::
$here = getcwd();

require Cwd;          # make Cwd:: accessible
$here = getcwd();     # oops! no main::getcwd()

```

In general, use `Module ()` is recommended over `require Module`, because it determines module availability at compile time, not in the middle of your program's execution. An exception would be if two modules each tried to use each other, and each also called a function from that other module. In that case, it's easy to use `requires` instead.

Perl packages may be nested inside other package names, so we can have package names containing `::`. But if we used that package name directly as a filename it would make for unwieldy or impossible filenames on some systems. Therefore, if a module's name is, say, `Text::Soundex`, then its definition is actually found in the library file *Text/Soundex.pm*.

Perl modules always have a *.pm* file, but there may also be dynamically linked executables (often ending in *.so*) or autoloading subroutine definitions (often ending in *.al*) associated with the module. If so, these will be entirely transparent to the user of the module. It is the responsibility of the *.pm* file to load (or arrange to autoload) any additional functionality. For example, although the `POSIX` module happens to do both dynamic loading and autoloading, the user can say just use `POSIX` to get it all.

Making your module threadsafe

Perl has since 5.6.0 support for a new type of threads called interpreter threads. These threads can be used explicitly and implicitly.

Threads work by cloning the data tree so that no data is shared between different threads. These threads can be used using the `threads` module or by doing `fork()` on win32 (fake `fork()` support). When a thread is cloned all Perl data is cloned, however non-Perl data cannot be cloned automatically. Perl after 5.7.2 has support for the `CLONE` special subroutine. In `CLONE` you can do whatever you need to do, like for example handle the cloning of non-Perl data, if necessary. `CLONE` will be executed once for every package that has it defined (or inherits it). It will be called in the context of the new thread, so all modifications are made in the new area.

If you want to `CLONE` all objects you will need to keep track of them per package. This is simply done using a hash and `Scalar::Util::weaken()`.

SEE ALSO

See [perlmodlib](#) for general style issues related to building Perl modules and classes, as well as descriptions of the standard library and CPAN, [Exporter](#) for how Perl's standard import/export mechanism works, [perltoot](#) and [perltooc](#) for an in-depth tutorial on creating classes, [perlobj](#) for a hard-core reference document on objects, [perlsub](#) for an explanation of functions and scoping, and [perlxsut](#) and [perlguts](#) for more information on writing extension modules.

NAME

perlmodinstall – Installing CPAN Modules

DESCRIPTION

You can think of a module as the fundamental unit of reusable Perl code; see [perlmod](#) for details. Whenever anyone creates a chunk of Perl code that they think will be useful to the world, they register as a Perl developer at <http://www.cpan.org/modules/04pause.html> so that they can then upload their code to the CPAN. The CPAN is the Comprehensive Perl Archive Network and can be accessed at <http://www.cpan.org/>, and searched at <http://search.cpan.org/>.

This documentation is for people who want to download CPAN modules and install them on their own computer.

PREAMBLE

First, are you sure that the module isn't already on your system? Try `perl -MFoo -e 1`. (Replace "Foo" with the name of the module; for instance, `perl -MCGI::Carp -e 1`.)

If you don't see an error message, you have the module. (If you do see an error message, it's still possible you have the module, but that it's not in your path, which you can display with `perl -e "print qq(@INC)".`) For the remainder of this document, we'll assume that you really honestly truly lack an installed module, but have found it on the CPAN.

So now you have a file ending in `.tar.gz` (or, less often, `.zip`). You know there's a tasty module inside. There are four steps you must now take:

DECOMPRESS the file

UNPACK the file into a directory

BUILD the module (sometimes unnecessary)

INSTALL the module.

Here's how to perform each step for each operating system. This is <not a substitute for reading the README and INSTALL files that might have come with your module!

Also note that these instructions are tailored for installing the module into your system's repository of Perl modules — but you can install modules into any directory you wish. For instance, where I say `perl Makefile.PL`, you can substitute `perl Makefile.PL PREFIX=/my/perl_directory` to install the modules into `/my/perl_directory`. Then you can use the modules from your Perl programs with `use lib "/my/perl_directory/lib/site_perl";` or sometimes just use `"/my/perl_directory";`. If you're on a system that requires superuser/root access to install modules into the directories you see when you type `perl -e "print qq(@INC)"`, you'll want to install them into a local directory (such as your home directory) and use this approach.

- **If you're on a Unix or Linux system,**

You can use Andreas Koenig's CPAN module (<http://www.cpan.org/modules/by-module/CPAN>) to automate the following steps, from DECOMPRESS through INSTALL.

A. DECOMPRESS

Decompress the file with `gzip -d yourmodule.tar.gz`

You can get `gzip` from <ftp://prep.ai.mit.edu/pub/gnu/>

Or, you can combine this step with the next to save disk space:

```
gzip -dc yourmodule.tar.gz | tar -xof -
```

B. UNPACK

Unpack the result with `tar -xof yourmodule.tar`

C. BUILD

Go into the newly-created directory and type:

```
perl Makefile.PL
make
make test
```

or

```
perl Makefile.PL PREFIX=/my/perl_directory
```

to install it locally. (Remember that if you do this, you'll have to put `use lib "/my/perl_directory";` near the top of the program that is to use this module.

D. INSTALL

While still in that directory, type:

```
make install
```

Make sure you have the appropriate permissions to install the module in your Perl 5 library directory. Often, you'll need to be root.

That's all you need to do on Unix systems with dynamic linking. Most Unix systems have dynamic linking — if yours doesn't, or if for another reason you have a statically-linked perl, **and** the module requires compilation, you'll need to build a new Perl binary that includes the module. Again, you'll probably need to be root.

- **If you're running ActivePerl (Win95/98/2K/NT/XP, Linux, Solaris)**

First, type ppm from a shell and see whether ActiveState's PPM repository has your module. If so, you can install it with ppm and you won't have to bother with any of the other steps here. You might be able to use the CPAN instructions from the "Unix or Linux" section above as well; give it a try. Otherwise, you'll have to follow the steps below.

A. DECOMPRESS

You can use the shareware Winzip (<http://www.winzip.com>) to decompress and unpack modules.

B. UNPACK

If you used WinZip, this was already done for you.

C. BUILD

Does the module require compilation (i.e. does it have files that end in .xs, .c, .h, .y, .cc, .cxx, or .C)? If it doesn't, go to INSTALL. If it does, life is now officially tough for you, because you have to compile the module yourself — no easy feat on Windows. You'll need the nmake utility, available at <ftp://ftp.microsoft.com/Softlib/MSLFILES/nmake15.exe>

D. INSTALL

Copy the module into your Perl's *lib* directory. That'll be one of the directories you see when you type

```
perl -e 'print "@INC"'
```

- **If you're using a Macintosh,**

A. DECOMPRESS

First, make sure you have the latest **cpan-mac** distribution (<http://www.cpan.org/authors/id/CNANDOR/>), which has utilities for doing all of the steps. Read the cpan-mac directions carefully and install it. If you choose not to use cpan-mac for some reason, there are alternatives listed here.

After installing cpan-mac, drop the module archive on the **untarzipme** droplet, which will

decompress and unpack for you.

Or, you can either use the shareware **StuffIt Expander** program (<http://www.aladdinsys.com/expander/>) in combination with **DropStuff with Expander Enhancer** (<http://www.aladdinsys.com/dropstuff/>) or the freeware **MacGzip** program (<http://persephone.cps.unizar.es/general/gente/spd/gzip/gzip.html>).

B. UNPACK

If you're using `untarzipme` or `StuffIt`, the archive should be extracted now. **Or**, you can use the freeware **suntar** or **Tar** (<http://hyperarchive.lcs.mit.edu/HyperArchive/Archive/cmp/>).

C. BUILD

Check the contents of the distribution. Read the module's documentation, looking for reasons why you might have trouble using it with MacPerl. Look for `.xs` and `.c` files, which normally denote that the distribution must be compiled, and you cannot install it "out of the box." (See *"PORTABILITY"*.)

If a module does not work on MacPerl but should, or needs to be compiled, see if the module exists already as a port on the MacPerl Module Porters site (<http://pudge.net/mmp/>). For more information on doing XS with MacPerl yourself, see Arved Sandstrom's XS tutorial (<http://macperl.com/depts/Tutorials/>), and then consider uploading your binary to the CPAN and registering it on the MMP site.

D. INSTALL

If you are using `cpan-mac`, just drop the folder on the **installme** droplet, and use the module.

Or, if you aren't using `cpan-mac`, do some manual labor.

Make sure the newlines for the modules are in Mac format, not Unix format. If they are not then you might have decompressed them incorrectly. Check your decompression and unpacking utilities settings to make sure they are translating text files properly.

As a last resort, you can use the perl one-liner:

```
perl -i.bak -pe 's/(?:\015)?\012/\015/g' <filenames>
```

on the source files.

Then move the files (probably just the `.pm` files, though there may be some additional ones, too; check the module documentation) to their final destination: This will most likely be in `$ENV{MACPERL}site_lib:` (i.e., `HD:MacPerl folder:site_lib:`). You can add new paths to the default `@INC` in the Preferences menu item in the MacPerl application (`$ENV{MACPERL}site_lib:` is added automatically). Create whatever directory structures are required (i.e., for `Some::Module`, create `$ENV{MACPERL}site_lib:Some:` and put `Module.pm` in that directory).

Then run the following script (or something like it):

```
#!/perl -w
use AutoSplit;
my $dir = "${MACPERL}site_perl";
autosplit("$dir:Some:Module.pm", "$dir:auto", 0, 1, 1);
```

- **If you're on the DJGPP port of DOS,**

A. DECOMPRESS

`djtarx` (<ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2/>) will both uncompress and unpack.

B. UNPACK

See above.

C. BUILD

Go into the newly-created directory and type:

```
perl Makefile.PL
make
make test
```

You will need the packages mentioned in *README.dos* in the Perl distribution.

D. INSTALL

While still in that directory, type:

```
make install
```

You will need the packages mentioned in *README.dos* in the Perl distribution.

- **If you're on OS/2,**

Get the EMX development suite and gzip/tar, from either Hobbes (<http://hobbes.nmsu.edu>) or Leo (<http://www.leo.org>), and then follow the instructions for Unix.

- **If you're on VMS,**

When downloading from CPAN, save your file with a .tgz extension instead of .tar.gz. All other periods in the filename should be replaced with underscores. For example, Your-Module-1.33.tar.gz should be downloaded as Your-Module-1_33.tgz.

A. DECOMPRESS

Type

```
gzip -d Your-Module.tgz
```

or, for zipped modules, type

```
unzip Your-Module.zip
```

Executables for gzip, zip, and VMStar:

```
http://www.openvms.digital.com/freeware/
http://www.crinoid.com/utills/
```

and their source code:

```
http://www.fsf.org/order/ftp.html
```

Note that GNU's gzip/gunzip is not the same as Info-ZIP's zip/unzip package. The former is a simple compression tool; the latter permits creation of multi-file archives.

B. UNPACK

If you're using VMStar:

```
VMStar xf Your-Module.tar
```

Or, if you're fond of VMS command syntax:

```
tar/extract/verbose Your_Module.tar
```

C. BUILD

Make sure you have MMS (from Digital) or the freeware MMK (available from MadGoat at <http://www.madgoat.com>). Then type this to create the DESCRIP.MMS for the module:

```
perl Makefile.PL
```

Now you're ready to build:

```
mms
mms test
```

Substitute mmk for mms above if you're using MMK.

D. INSTALL

Type

```
mms install
```

Substitute mmk for mms above if you're using MMK.

- **If you're on MVS,**

Introduce the *.tar.gz* file into an HFS as binary; don't translate from ASCII to EBCDIC.

A. DECOMPRESS

Decompress the file with `gzip -d yourmodule.tar.gz`

You can get gzip from <http://www.s390.ibm.com/products/oe/bpxqp1.html>

B. UNPACK

Unpack the result with

```
pax -o to=IBM-1047,from=ISO8859-1 -r < yourmodule.tar
```

The BUILD and INSTALL steps are identical to those for Unix. Some modules generate Makefiles that work better with GNU make, which is available from <http://www.mks.com/s390/gnu/>

PORTABILITY

Note that not all modules will work with on all platforms. See [perlport](#) for more information on portability issues. Read the documentation to see if the module will work on your system. There are basically three categories of modules that will not work "out of the box" with all platforms (with some possibility of overlap):

- **Those that should, but don't.** These need to be fixed; consider contacting the author and possibly writing a patch.
- **Those that need to be compiled, where the target platform doesn't have compilers readily available.** (These modules contain *.xs* or *.c* files, usually.) You might be able to find existing binaries on the CPAN or elsewhere, or you might want to try getting compilers and building it yourself, and then release the binary for other poor souls to use.
- **Those that are targeted at a specific platform.** (Such as the Win32:: modules.) If the module is targeted specifically at a platform other than yours, you're out of luck, most likely.

Check the CPAN Testers if a module should work with your platform but it doesn't behave as you'd expect, or you aren't sure whether or not a module will work under your platform. If the module you want isn't listed there, you can test it yourself and let CPAN Testers know, you can join CPAN Testers, or you can request it be tested.

<http://testers.cpan.org/>

HEY

If you have any suggested changes for this page, let me know. Please don't send me mail asking for help on how to install your modules. There are too many modules, and too few Orwants, for me to be able to answer or even acknowledge all your questions. Contact the module author instead, or post to [comp.lang.perl.modules](#), or ask someone familiar with Perl on your operating system.

AUTHOR

Jon Orwant

orwant@tpj.com

The Perl Journal, <http://tpj.com>

with invaluable help from Chris Nandor, and valuable help from Brandon Allbery, Charles Bailey, Graham Barr, Dominic Dunlop, Jarkko Hietaniemi, Ben Holzman, Tom Horsley, Nick Ing-Simmons, Tuomas J. Lukka, Laszlo Molnar, Alan Olsen, Peter Prymmer, Gurusamy Sarathy, Christoph Spalinger, Dan Sugalski, Larry Virden, and Ilya Zakharevich.

First version July 22, 1998; last revised November 21, 2001.

COPYRIGHT

Copyright (C) 1998, 2002 Jon Orwant. All Rights Reserved.

Permission is granted to make and distribute verbatim copies of this documentation provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that they are marked clearly as modified versions, that the authors' names and title are unchanged (though subtitles and additional authors' names may be added), and that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions.

NAME

perlmodlib – constructing new Perl modules and finding existing ones

DESCRIPTION**THE PERL MODULE LIBRARY**

Many modules are included the Perl distribution. These are described below, and all end in *.pm*. You may discover compiled library file (usually ending in *.so*) or small pieces of modules to be autoloading (ending in *.al*); these were automatically generated by the installation process. You may also discover files in the library directory that end in either *.pl* or *.ph*. These are old libraries supplied so that old programs that use them still run. The *.pl* files will all eventually be converted into standard modules, and the *.ph* files made by **h2ph** will probably end up as extension modules made by **h2xs**. (Some *.ph* values may already be available through the POSIX, Erno, or Fcntl modules.) The **pl2pm** file in the distribution may help in your conversion, but it's just a mechanical process and therefore far from bulletproof.

Pragmatic Modules

They work somewhat like compiler directives (pragmata) in that they tend to affect the compilation of your program, and thus will usually work well only when used within a `use`, or `no`. Most of these are lexically scoped, so an inner BLOCK may countermand them by saying:

```
no integer;
no strict 'refs';
no warnings;
```

which lasts until the end of that BLOCK.

Some pragmas are lexically scoped—typically those that affect the `$_H` hints variable. Others affect the current package instead, like `use vars` and `use subs`, which allow you to predeclare a variables or subroutines within a particular *file* rather than just a block. Such declarations are effective for the entire file for which they were declared. You cannot rescind them with `no vars` or `no subs`.

The following pragmas are defined (and have their own documentation).

attributes	Get/set subroutine or variable attributes
attrs	Set/get attributes of a subroutine (deprecated)
autouse	Postpone load of modules until a function is used
base	Establish IS–A relationship with base class at compile time
bigint	Transparent big integer support for Perl
bignum	Transparent BigNumber support for Perl
bigrat	Transparent BigNumber/BigRational support for Perl
blib	Use MakeMaker's uninstalled version of a package
bytes	Force byte semantics rather than character semantics
charnames	Define character names for <code>\N{named}</code> string literal escapes
constant	Declare constants
diagnostics	Perl compiler pragma to force verbose warning diagnostics
encoding	Allows you to write your script in non-ascii or non-utf8
fields	Compile-time class fields
filetest	Control the filetest permission operators

<code>if</code>	use a Perl module if a condition holds
<code>integer</code>	Use integer arithmetic instead of floating point
<code>less</code>	Request less of something from the compiler
<code>locale</code>	Use and avoid POSIX locales for built-in operations
<code>open</code>	Set default PerlIO layers for input and output
<code>ops</code>	Restrict unsafe operations when compiling
<code>overload</code>	Package for overloading perl operations
<code>re</code>	Alter regular expression behaviour
<code>sigtrap</code>	Enable simple signal handling
<code>sort</code>	Control <code>sort()</code> behaviour
<code>strict</code>	Restrict unsafe constructs
<code>subs</code>	Predeclare sub names
<code>threads</code>	Perl extension allowing use of interpreter based threads from perl
<code>utf8</code>	Enable/disable UTF-8 (or UTF-EBCDIC) in source code
<code>vars</code>	Predeclare global variable names (obsolete)
<code>vmsish</code>	Control VMS-specific language features
<code>warnings</code>	Control optional warnings
<code>warnings::register</code>	Warnings import function

Standard Modules

Standard, bundled modules are all expected to behave in a well-defined manner with respect to namespace pollution because they use the Exporter module. See their own documentation for details.

`AnyDBM_File` Provide framework for multiple DBMs

Attribute::Handlers

Simpler definition of attribute handlers

`AutoLoader` Load subroutines only on demand

`AutoSplit` Split a package for autoloading

`B` The Perl Compiler

`B::Asmdata` Autogenerated data about Perl ops, used to generate bytecode

`B::Assembler` Assemble Perl bytecode

`B::Block` Walk basic blocks

`B::Bytecode` Perl compiler's bytecode backend

`B::C` Perl compiler's C backend

`B::CC` Perl compiler's optimized C translation backend

`B::Concise` Walk Perl syntax tree, printing concise info about ops

`B::Debug` Walk Perl syntax tree, printing debug info about ops

B::Deparse	Perl compiler backend to produce perl code
B::Disassembler	Disassemble Perl bytecode
B::Lint	Perl lint
B::Showlex	Show lexical variables used in functions or files
B::Stackobj	Helper module for CC backend
B::Stash	Show what stashes are loaded
B::Terse	Walk Perl syntax tree, printing terse info about ops
B::Xref	Generates cross reference reports for Perl programs
Benchmark	Benchmark running times of Perl code
ByteLoader	Load byte compiled perl code
CGI	Simple Common Gateway Interface Class
CGI::Apache	Backward compatibility module for CGI.pm
CGI::Carp	CGI routines for writing to the HTTPD (or other) error log
CGI::Cookie	Interface to Netscape Cookies
CGI::Fast	CGI Interface for Fast CGI
CGI::Pretty	Module to produce nicely formatted HTML code
CGI::Push	Simple Interface to Server Push
CGI::Switch	Backward compatibility module for defunct CGI::Switch
CGI::Util	Internal utilities used by CGI module
CPAN	Query, download and build perl modules from CPAN sites
CPAN::FirstTime	Utility for CPAN::Config file Initialization
CPAN::Nox	Wrapper around CPAN.pm without using any XS module
Carp	Warn of errors (from perspective of caller)
Carp::Heavy	No user serviceable parts inside
Class::ISA	Report the search path for a class's ISA tree
Class::Struct	Declare struct-like datatypes as Perl classes
Config	Access Perl configuration information
Cwd	Get pathname of current working directory
DB	Programmatic interface to the Perl debugging API (draft, subject to
DB_File	Perl5 access to Berkeley DB version 1.x
Devel::SelfStubber	Generate stubs for a SelfLoading module
Digest	Modules that calculate message digests

DirHandle	Supply object methods for directory handles
Dumpvalue	Provides screen dump of Perl data.
Encode	Character encodings
English	Use nice English (or awk) names for ugly punctuation variables
Env	Perl module that imports environment variables as scalars or arrays
Errno	System errno constants
Exporter	Implements default import method for modules
Exporter::Heavy	Exporter guts
ExtUtils::Command	Utilities to replace common UNIX commands in Makefiles etc.
ExtUtils::Command::MM	Commands for the MM's to use in Makefiles
ExtUtils::Constant	Generate XS code to import C header constants
ExtUtils::Embed	Utilities for embedding Perl in C/C++ applications
ExtUtils::Install	Install files from here to there
ExtUtils::Installed	Inventory management of installed modules
ExtUtils::Liblist	Determine libraries to use and how to use them
ExtUtils::MM	OS adjusted ExtUtils::MakeMaker subclass
ExtUtils::MM_Any	Platform agnostic MM methods
ExtUtils::MM_BeOS	Methods to override UN*X behaviour in ExtUtils::MakeMaker
ExtUtils::MM_Cygwin	Methods to override UN*X behaviour in ExtUtils::MakeMaker
ExtUtils::MM_DOS	DOS specific subclass of ExtUtils::MM_Unix
ExtUtils::MM_MacOS	Methods to override UN*X behaviour in ExtUtils::MakeMaker
ExtUtils::MM_NW5	Methods to override UN*X behaviour in ExtUtils::MakeMaker
ExtUtils::MM_OS2	Methods to override UN*X behaviour in ExtUtils::MakeMaker
ExtUtils::MM_UWIN	U/WIN specific subclass of ExtUtils::MM_Unix

ExtUtils::MM_Unix
Methods used by ExtUtils::MakeMaker

ExtUtils::MM_VMS
Methods to override UN*X behaviour in ExtUtils::MakeMaker

ExtUtils::MM_Win32
Methods to override UN*X behaviour in ExtUtils::MakeMaker

ExtUtils::MM_Win95
Method to customize MakeMaker for Win9X

ExtUtils::MY ExtUtils::MakeMaker subclass for customization

ExtUtils::MakeMaker
Create an extension Makefile

ExtUtils::Manifest
Utilities to write and check a MANIFEST file

ExtUtils::Mkbootstrap
Make a bootstrap file for use by DynaLoader

ExtUtils::Mksymlists
Write linker options files for dynamic extension

ExtUtils::Packlist
Manage .packlist files

ExtUtils::testlib Add blib/* directories to @INC

Fatal Replace functions with equivalents which succeed or die

Fcntl Load the C Fcntl.h defines

File::Basename
Split a pathname into pieces

File::CheckTree
Run many filetest checks on a tree

File::Compare Compare files or filehandles

File::Copy Copy files or filehandles

File::DosGlob DOS like globbing and then some

File::Find Traverse a directory tree.

File::Path Create or remove directory trees

File::Spec Portably perform operations on file names

File::Spec::Cygwin
Methods for Cygwin file specs

File::Spec::Epoc
Methods for Epoc file specs

File::Spec::Functions
Portably perform operations on file names

`File::Spec::Mac`
File::Spec for Mac OS (Classic)

`File::Spec::OS2`
Methods for OS/2 file specs

`File::Spec::Unix`
File::Spec for Unix, base for other File::Spec modules

`File::Spec::VMS`
Methods for VMS file specs

`File::Spec::Win32`
Methods for Win32 file specs

`File::Temp` Return name and handle of a temporary file safely

`File::stat` By-name interface to Perl's built-in `stat()` functions

`FileCache` Keep more files open than the system permits

`FileHandle` Supply object methods for filehandles

`Filter::Simple` Simplified source filtering

`FindBin` Locate directory of original perl script

`Getopt::Long` Extended processing of command line options

`Getopt::Std` Process single-character switches with switch clustering

`Hash::Util` A selection of general-utility hash subroutines

`I18N::Collate` Compare 8-bit scalar data according to the current locale

`I18N::LangTags`
Functions for dealing with RFC3066-style language tags

`I18N::LangTags::List`
Tags and names for human languages

`IO` Load various IO modules

`IPC::Open2` Open a process for both reading and writing

`IPC::Open3` Open a process for reading, writing, and error handling

`Locale::Constants`
Constants for Locale codes

`Locale::Country`
ISO codes for country identification (ISO 3166)

`Locale::Currency`
ISO three letter codes for currency identification (ISO 4217)

`Locale::Language`
ISO two letter codes for language identification (ISO 639)

`Locale::Maketext`
Framework for localization

Locale::Maketext::TPJ13	Article about software localization
Locale::Script	ISO codes for script identification (ISO 15924)
Math::BigFloat	Arbitrary size floating point math package
Math::BigInt	Arbitrary size integer math package
Math::BigInt::Calc	Pure Perl module to support Math::BigInt
Math::BigRat	Arbitrarily big rationals
Math::Complex	Complex numbers and associated mathematical functions
Math::Trig	Trigonometric functions
Memoize	Make functions faster by trading space for time
Memoize::AnyDBM_File	Glue to provide EXISTS for AnyDBM_File for Storable use
Memoize::Expire	Plug-in module for automatic expiration of memoized values
Memoize::ExpireFile	Test for Memoize expiration semantics
Memoize::ExpireTest	Test for Memoize expiration semantics
Memoize::NDBM_File	Glue to provide EXISTS for NDBM_File for Storable use
Memoize::SDBM_File	Glue to provide EXISTS for SDBM_File for Storable use
Memoize::Storable	Store Memoized data in Storable database
NDBM_File	Tied access to ndbm files
NEXT	Provide a pseudo-class NEXT that allows method redispatch
Net::Cmd	Network Command class (as used by FTP, SMTP etc)
Net::Config	Local configuration data for libnet
Net::Domain	Attempt to evaluate the current host's internet name and domain
Net::FTP	FTP Client class
Net::NNTP	NNTP Client class
Net::Netrc	OO interface to users netrc file
Net::POP3	Post Office Protocol 3 Client class (RFC1939)
Net::Ping	Check a remote host for reachability
Net::SMTP	Simple Mail Transfer Protocol Client

Net::Time	Time and daytime network client interface
Net::hostent	By-name interface to Perl's built-in <code>gethost*</code> () functions
Net::libnetFAQ	Libnet Frequently Asked Questions
Net::netent	By-name interface to Perl's built-in <code>getnet*</code> () functions
Net::protoent	By-name interface to Perl's built-in <code>getproto*</code> () functions
Net::servent	By-name interface to Perl's built-in <code>getserv*</code> () functions
O	Generic interface to Perl Compiler backends
ODBM_File	Tied access to odbm files
Opcode	Disable named opcodes when compiling perl code
POSIX	Perl interface to IEEE Std 1003.1
PerlIO	On demand loader for PerlIO layers and root of <code>PerlIO::*</code> name space
PerlIO::via::QuotedPrint	PerlIO layer for quoted-printable strings
Pod::Checker	Check pod documents for syntax errors
Pod::Find	Find POD documents in directory trees
Pod::Functions	Group Perl's functions a la <code>perlfunc.pod</code>
Pod::Html	Module to convert pod files to HTML
Pod::InputObjects	Objects representing POD input paragraphs, commands, etc.
Pod::LaTeX	Convert Pod data to formatted Latex
Pod::Man	Convert POD data to formatted <code>*roff</code> input
Pod::ParseLink	Parse an <code>L<></code> formatting code in POD text
Pod::ParseUtils	Helpers for POD parsing and conversion
Pod::Parser	Base class for creating POD filters and translators
Pod::Plainer	Perl extension for converting Pod to old style Pod.
Pod::Select	Extract selected sections of POD from input
Pod::Text	Convert POD data to formatted ASCII text
Pod::Text::Color	Convert POD data to formatted color ASCII text
Pod::Text::Overstrike	Convert POD data to formatted overstrike text
Pod::Text::Termcap	Convert POD data to ASCII text with format escapes
Pod::Usage	Print a usage message from embedded pod documentation

SDBM_File	Tied access to sdbm files
Safe	Compile and execute code in restricted compartments
Search::Dict	Search for key in dictionary file
SelectSaver	Save and restore selected file handle
SelfLoader	Load functions only on demand
Shell	Run shell commands transparently within perl
Socket	Load the C socket.h defines and structure manipulators
Storable	Persistence for Perl data structures
Switch	A switch statement for Perl
Symbol	Manipulate Perl symbols and their names
Term::ANSIColor	Color screen output using ANSI escape sequences
Term::Cap	Perl termcap interface
Term::Complete	Perl word completion module
Term::ReadLine	Perl interface to various readline packages. If
Test	Provides a simple framework for writing test scripts
Test::Builder	Backend for building test libraries
Test::Harness	Run perl standard test scripts with statistics
Test::Harness::Assert	Simple assert
Test::Harness::Iterator	Internal Test::Harness Iterator
Test::Harness::Straps	Detailed analysis of test results
Test::More	Yet another framework for writing test scripts
Test::Simple	Basic utilities for writing tests.
Test::Tutorial	A tutorial about writing really basic tests
Text::Abbrev	Create an abbreviation table from a list
Text::Balanced	Extract delimited text sequences from strings.
Text::ParseWords	Parse text into an array of tokens or array of arrays
Text::Soundex	Implementation of the Soundex Algorithm as Described by Knuth
Text::Tabs	Expand and unexpand tabs per the unix expand(1) and unexpand(1)
Text::Wrap	Line wrapping to form simple paragraphs

Thread	Manipulate threads in Perl (for old code only)
Thread::Queue	Thread-safe queues
Thread::Semaphore	Thread-safe semaphores
Tie::Array	Base class for tied arrays
Tie::File	Access the lines of a disk file via a Perl array
Tie::Handle	Base class definitions for tied handles
Tie::Hash	Base class definitions for tied hashes
Tie::Memoize	Add data to hash when needed
Tie::RefHash	Use references as hash keys
Tie::Scalar	Base class definitions for tied scalars
Tie::SubstrHash	Fixed-table-size, fixed-key-length hashing
Time::Local	Efficiently compute time from local and GMT time
Time::gmtime	By-name interface to Perl's built-in <code>gmtime()</code> function
Time::localtime	By-name interface to Perl's built-in <code>localtime()</code> function
Time::tm	Internal object used by <code>Time::gmtime</code> and <code>Time::localtime</code>
UNIVERSAL	Base class for ALL classes (blessed references)
Unicode::Collate	Unicode Collation Algorithm
Unicode::UCD	Unicode character database
User::grent	By-name interface to Perl's built-in <code>getgr*</code> functions
User::pwent	By-name interface to Perl's built-in <code>getpw*</code> functions
Win32	Interfaces to some Win32 API Functions

To find out *all* modules installed on your system, including those without documentation or outside the standard release, just do this:

```
% find `perl -e 'print "@INC"'\` -name '*.pm' -print
```

They should all have their own documentation installed and accessible via your system `man(1)` command. If you do not have a **find** program, you can use the Perl **find2perl** program instead, which generates Perl code as output you can run through perl. If you have a **man** program but it doesn't find your modules, you'll have to fix your manpath. See [perl](#) for details. If you have no system **man** command, you might try the **perldoc** program.

Extension Modules

Extension modules are written in C (or a mix of Perl and C). They are usually dynamically loaded into Perl if and when you need them, but may also be linked in statically. Supported extension modules include Socket, Fcntl, and POSIX.

Many popular C extension modules do not come bundled (at least, not completely) due to their sizes, volatility, or simply lack of time for adequate testing and configuration across the multitude of platforms on

which Perl was beta-tested. You are encouraged to look for them on CPAN (described below), or using web search engines like Alta Vista or Deja News.

CPAN

CPAN stands for Comprehensive Perl Archive Network; it's a globally replicated trove of Perl materials, including documentation, style guides, tricks and traps, alternate ports to non-Unix systems and occasional binary distributions for these. Search engines for CPAN can be found at <http://www.cpan.org/>

Most importantly, CPAN includes around a thousand unbundled modules, some of which require a C compiler to build. Major categories of modules are:

- Language Extensions and Documentation Tools
- Development Support
- Operating System Interfaces
- Networking, Device Control (modems) and InterProcess Communication
- Data Types and Data Type Utilities
- Database Interfaces
- User Interfaces
- Interfaces to / Emulations of Other Programming Languages
- File Names, File Systems and File Locking (see also File Handles)
- String Processing, Language Text Processing, Parsing, and Searching
- Option, Argument, Parameter, and Configuration File Processing
- Internationalization and Locale
- Authentication, Security, and Encryption
- World Wide Web, HTML, HTTP, CGI, MIME
- Server and Daemon Utilities
- Archiving and Compression
- Images, Pixmap and Bitmap Manipulation, Drawing, and Graphing
- Mail and Usenet News
- Control Flow Utilities (callbacks and exceptions etc)
- File Handle and Input/Output Stream Utilities
- Miscellaneous Modules

The list of the registered CPAN sites as of this writing follows. Please note that the sorting order is alphabetical on fields:

```
Continent
|
|—Country
|
|—[state/province]
|
|—ftp
|
|—[http]
```

and thus the North American servers happen to be listed between the European and the South American

sites.

You should try to choose one close to you.

Africa

South Africa

```
ftp://ftp.is.co.za/programming/perl/CPAN/  
ftp://ftp.mweb.co.za/pub/mirrors/cpan/  
ftp://ftp.saix.net/pub/CPAN/  
ftp://ftp.sun.ac.za/CPAN/CPAN/
```

Asia

China

```
ftp://freesoft.cei.gov.cn/pub/languages/perl/CPAN/  
http://www2.linuxforum.net/mirror/CPAN/  
http://cpan.shellhung.org/  
ftp://ftp.shellhung.org/pub/CPAN
```

India

```
http://cpan.in.freeos.com  
ftp://cpan.in.freeos.com/pub/CPAN/
```

Indonesia

```
http://cpan.itb.web.id/  
ftp://mirrors.piksi.itb.ac.id/CPAN/  
http://cpan.cbn.net.id/  
ftp://ftp.cbn.net.id/mirror/CPAN  
http://CPAN.mweb.co.id/  
ftp://ftp.mweb.co.id/pub/languages/perl/CPAN/
```

Israel

```
http://www.iglu.org.il:/pub/CPAN/  
ftp://ftp.iglu.org.il/pub/CPAN/  
http://cpan.lerner.co.il/  
http://bioinfo.weizmann.ac.il/pub/software/perl/CPAN/  
ftp://bioinfo.weizmann.ac.il/pub/software/perl/CPAN/
```

Japan

```
ftp://ftp.u-aizu.ac.jp/pub/CPAN  
ftp://ftp.kddlabs.co.jp/CPAN/  
http://mirror.nucba.ac.jp/mirror/Perl/  
ftp://mirror.nucba.ac.jp/mirror/Perl/  
ftp://ftp.meisei-u.ac.jp/pub/CPAN/  
ftp://ftp.ayamura.org/pub/CPAN/  
ftp://ftp.jaist.ac.jp/pub/lang/perl/CPAN/  
ftp://ftp.dti.ad.jp/pub/lang/CPAN/  
ftp://ftp.ring.gr.jp/pub/lang/perl/CPAN/
```

Korea

```
http://mirror.Mazic.org/pub/CPAN  
ftp://mirror.Mazic.org/pub/CPAN
```

Philippines

```
http://www.adzu.edu.ph/CPAN
```

Russian Federation

<http://cpan.tomsk.ru>
<ftp://cpan.tomsk.ru/pub/CPAN>

Saudi Arabia

<ftp://ftp.isu.net.sa/pub/CPAN/>

Singapore

<http://cpan.hjc.edu.sg>
<http://mirror.averse.net/pub/CPAN>
<ftp://mirror.averse.net/pub/CPAN>

South Korea

<http://CPAN.bora.net/>
<ftp://ftp.bora.net/pub/CPAN/>
<http://ftp.kornet.net/pub/CPAN/>
<ftp://ftp.kornet.net/pub/CPAN/>
<ftp://ftp.nuri.net/pub/CPAN/>
<http://ftp.xgate.co.kr/cpan/>
ftp://ftp.xgate.co.kr/pub/mirror/CPAN

Taiwan

<ftp://ftp.nctu.edu.tw/UNIX/perl/CPAN>
<ftp://ftp.ee.ncku.edu.tw/pub/perl/CPAN/>
<ftp://ftp1.sinica.edu.tw/pub1/perl/CPAN/>
<http://ftp.tku.edu.tw/pub/CPAN/>
<ftp://ftp.tku.edu.tw/pub/CPAN/>

Thailand

<ftp://ftp.loxinfo.co.th/pub/cpan/>
<ftp://ftp.cs.riubon.ac.th/pub/mirrors/CPAN/>

Central America

Costa Rica

<ftp://ftp.linux.co.cr/mirrors/CPAN/>
<http://ftp.ucr.ac.cr/Unix/CPAN/>
<ftp://ftp.ucr.ac.cr/pub/Unix/CPAN/>

Europe

Austria

<ftp://ftp.tuwien.ac.at/pub/CPAN/>

Belgium

<http://ftp.easynet.be/pub/CPAN/>
<ftp://ftp.easynet.be/pub/CPAN/>
<http://cpan.skynet.be>
<ftp://ftp.skynet.be/pub/CPAN>
<ftp://ftp.kulnet.kuleuven.ac.be/pub/mirror/CPAN/>

Bulgaria

<http://cpan.lirex.net/>
<ftp://ftp.lirex.net/pub/mirrors/CPAN>

Croatia

<http://ftp.linux.hr/pub/CPAN/>
<ftp://ftp.linux.hr/pub/CPAN/>

Czech Republic

```
http://ftp.fi.muni.cz/pub/CPAN/  
ftp://ftp.fi.muni.cz/pub/CPAN/  
ftp://sunsite.mff.cuni.cz/MIRRORS/ftp.funet.fi/pub/language
```

Denmark

```
http://mirrors.sunsite.dk/cpan/  
ftp://sunsite.dk/mirrors/cpan/  
http://cpan.cybercity.dk  
http://www.cpan.dk/CPAN/  
ftp://www.cpan.dk/ftp.cpan.org/CPAN/
```

Estonia

```
ftp://ftp.ut.ee/pub/languages/perl/CPAN/
```

Finland

```
ftp://ftp.funet.fi/pub/languages/perl/CPAN/  
http://cpan.kpnqwest.fi/
```

France

```
http://ftp.u-paris10.fr/perl/CPAN  
ftp://ftp.u-paris10.fr/perl/CPAN  
http://cpan.mirrors.easynet.fr/  
ftp://cpan.mirrors.easynet.fr/pub/ftp.cpan.org/  
ftp://ftp.club-internet.fr/pub/perl/CPAN/  
http://fr.cpan.org/  
ftp://ftp.lip6.fr/pub/perl/CPAN/  
ftp://ftp.oleane.net/pub/mirrors/CPAN/  
ftp://ftp.pasteur.fr/pub/computing/CPAN/  
http://mir2.ovh.net/ftp.cpan.org  
ftp://mir1.ovh.net/ftp.cpan.org  
http://ftp.u-strasbg.fr/CPAN  
ftp://ftp.u-strasbg.fr/CPAN  
http://cpan.cict.fr/  
ftp://cpan.cict.fr/pub/CPAN/  
ftp://ftp.uvsq.fr/pub/perl/CPAN/
```

Germany

```
ftp://ftp.rz.ruhr-uni-bochum.de/pub/CPAN/  
ftp://ftp.freenet.de/pub/ftp.cpan.org/pub/CPAN/  
ftp://ftp.uni-erlangen.de/pub/source/CPAN/  
ftp://ftp-stud.fht-esslingen.de/pub/Mirrors/CPAN  
http://pandemonium.tiscali.de/pub/CPAN/  
ftp://pandemonium.tiscali.de/pub/CPAN/  
http://ftp.gwdg.de/pub/languages/perl/CPAN/  
ftp://ftp.gwdg.de/pub/languages/perl/CPAN/  
ftp://ftp.uni-hamburg.de/pub/soft/lang/perl/CPAN/  
ftp://ftp.leo.org/pub/CPAN/  
http://cpan.noris.de/  
ftp://cpan.noris.de/pub/CPAN/  
ftp://ftp.mpi-sb.mpg.de/pub/perl/CPAN/  
ftp://ftp.gmd.de/mirrors/CPAN/
```

Greece

```
ftp://ftp.acn.gr/pub/lang/perl/CPAN  
ftp://ftp.forthnet.gr/pub/languages/perl/CPAN
```

	<code>ftp://ftp.ntua.gr/pub/lang/perl/</code>
Hungary	<code>http://cpan.artifact.hu/</code> <code>ftp://cpan.artifact.hu/CPAN/</code> <code>http://ftp.kfki.hu/packages/perl/CPAN/</code> <code>ftp://ftp.kfki.hu/pub/packages/perl/CPAN/</code>
Iceland	<code>http://ftp.rhnet.is/pub/CPAN/</code> <code>ftp://ftp.rhnet.is/pub/CPAN/</code>
Ireland	<code>http://cpan.indigo.ie/</code> <code>ftp://cpan.indigo.ie/pub/CPAN/</code> <code>http://sunsite.compapp.dcu.ie/pub/perl/</code> <code>ftp://sunsite.compapp.dcu.ie/pub/perl/</code>
Italy	<code>http://cpan.nettuno.it/</code> <code>http://gusp.dyndns.org/CPAN/</code> <code>ftp://gusp.dyndns.org/pub/CPAN</code> <code>http://softcity.iol.it/cpan</code> <code>ftp://softcity.iol.it/pub/cpan</code> <code>ftp://ftp.unina.it/pub/Other/CPAN/CPAN/</code> <code>ftp://ftp.unipi.it/pub/mirror/perl/CPAN/</code> <code>ftp://cis.uniRoma2.it/CPAN/</code> <code>ftp://ftp.edisontel.it/pub/CPAN_Mirror/</code> <code>ftp://ftp.flashnet.it/pub/CPAN/</code>
Latvia	<code>http://kvin.lv/pub/CPAN/</code>
Lithuania	<code>ftp://ftp.unix.lt/pub/CPAN/</code>
Netherlands	<code>ftp://download.xs4all.nl/pub/mirror/CPAN/</code> <code>ftp://ftp.nl.uu.net/pub/CPAN/</code> <code>ftp://ftp.nluug.nl/pub/languages/perl/CPAN/</code> <code>http://cpan.cybercomm.nl/</code> <code>ftp://mirror.cybercomm.nl/pub/CPAN</code> <code>ftp://ftp.cpan.nl/pub/CPAN/</code> <code>http://ftp.easynet.nl/mirror/CPAN</code> <code>ftp://ftp.easynet.nl/mirror/CPAN</code> <code>http://archive.cs.uu.nl/mirror/CPAN/</code> <code>ftp://ftp.cs.uu.nl/mirror/CPAN/</code>
Norway	<code>ftp://ftp.uninett.no/pub/languages/perl/CPAN</code> <code>ftp://ftp.uit.no/pub/languages/perl/cpan/</code>
Poland	<code>ftp://ftp.pk.edu.pl/pub/lang/perl/CPAN/</code> <code>ftp://ftp.mega.net.pl/pub/mirrors/ftp.perl.com/</code> <code>ftp://ftp.man.torun.pl/pub/doc/CPAN/</code> <code>ftp://sunsite.icm.edu.pl/pub/CPAN/</code>

Portugal

```
ftp://ftp.ua.pt/pub/CPAN/  
ftp://perl.di.uminho.pt/pub/CPAN/  
http://cpan.dei.uc.pt/  
ftp://ftp.dei.uc.pt/pub/CPAN  
ftp://ftp.ist.utl.pt/pub/CPAN/  
http://cpan.ip.pt/  
ftp://cpan.ip.pt/pub/cpan/  
ftp://ftp.netc.pt/pub/CPAN/  
ftp://ftp.up.pt/pub/CPAN
```

Romania

```
ftp://ftp.kappa.ro/pub/mirrors/ftp.perl.org/pub/CPAN/  
ftp://ftp.dntis.ro/pub/cpan/  
ftp://ftp.dnttm.ro/pub/CPAN/  
ftp://ftp.lasting.ro/pub/CPAN  
ftp://ftp.timisoara.roedu.net/mirrors/CPAN/
```

Russia

```
ftp://ftp.chg.ru/pub/lang/perl/CPAN/  
http://cpan.rinet.ru/  
ftp://cpan.rinet.ru/pub/mirror/CPAN/  
ftp://ftp.aha.ru/pub/CPAN/  
http://cpan.sai.msu.ru/  
ftp://ftp.sai.msu.su/pub/lang/perl/CPAN/
```

Slovakia

```
http://ftp.cvt.stuba.sk/pub/CPAN/  
ftp://ftp.cvt.stuba.sk/pub/CPAN/
```

Slovenia

```
ftp://ftp.arnes.si/software/perl/CPAN/
```

Spain

```
http://cpan.imasd.elmundo.es/  
ftp://ftp.rediris.es/mirror/CPAN/  
ftp://ftp.etse.urv.es/pub/perl/
```

Sweden

```
http://ftp.du.se/CPAN/  
ftp://ftp.du.se/pub/CPAN/  
ftp://mirror.dataphone.se/pub/CPAN  
ftp://ftp.sunet.se/pub/lang/perl/CPAN/
```

Switzerland

```
ftp://ftp.danyk.ch/CPAN/  
ftp://sunsite.cnlab-switch.ch/mirror/CPAN/
```

Turkey

```
http://ftp.ulak.net.tr/perl/CPAN/  
ftp://ftp.ulak.net.tr/perl/CPAN  
ftp://sunsite.bilkent.edu.tr/pub/languages/CPAN/
```

Ukraine

```
http://cpan.org.ua/  
ftp://cpan.org.ua/  
ftp://ftp.perl.org.ua/pub/CPAN/
```

United Kingdom

<http://www.mirror.ac.uk/sites/ftp.funet.fi/pub/languages/perl/>
<ftp://ftp.mirror.ac.uk/sites/ftp.funet.fi/pub/languages/perl/>
<http://cpan.teleglobe.net/>
<ftp://cpan.teleglobe.net/pub/CPAN/>
<http://cpan.crazygreek.co.uk>
<ftp://ftp.demon.co.uk/pub/CPAN/>
<http://cpan.m.flirble.org/>
<ftp://ftp.flirble.org/pub/languages/perl/CPAN/>
<ftp://ftp.plig.org/pub/CPAN/>
<http://mirror.uklinux.net/CPAN/>
<ftp://mirror.uklinux.net/pub/CPAN/>
<http://cpan.mirrors.cloakerz.net/>
<ftp://ftp.cloakerz.net/pub/CPAN/>
<ftp://usit.shef.ac.uk/pub/packages/CPAN/>

North America

Alberta

<http://sunsite.ualberta.ca/pub/Mirror/CPAN/>
<ftp://sunsite.ualberta.ca/pub/Mirror/CPAN/>

Manitoba

<http://theoryx5.uwinnipeg.ca/pub/CPAN/>
<ftp://theoryx5.uwinnipeg.ca/pub/CPAN/>

Nova Scotia

<ftp://cpan.chebucto.ns.ca/pub/CPAN/>

Ontario

<ftp://ftp.crc.ca/pub/CPAN/>

Quebec

<http://cpan.mirror.smartworker.org/>

Mexico

<http://cpan.azc.uam.mx>
<ftp://cpan.azc.uam.mx/mirrors/CPAN>
<http://cpan.unam.mx/>
<ftp://cpan.unam.mx/pub/CPAN>
<http://www.msg.com.mx/CPAN/>
<ftp://ftp.msg.com.mx/pub/CPAN/>

United States

Alabama

<http://mirror.hiwaay.net/CPAN/>
<ftp://mirror.hiwaay.net/CPAN/>

California

<http://cpan.develooper.com/>
<http://www.cpan.org/>
<ftp://cpan.valueclick.com/pub/CPAN/>
<http://mirrors.gossamer-threads.com/CPAN>
<ftp://cpan.nas.nasa.gov/pub/perl/CPAN/>
<http://mirrors.kernel.org/cpan/>
<ftp://mirrors.kernel.org/pub/CPAN>

	http://cpan.digisle.net/ ftp://cpan.digisle.net/pub/CPAN http://www.perl.com/CPAN/ http://download.sourceforge.net/mirrors/CPAN/
Colorado	ftp://ftp.cs.colorado.edu/pub/perl/CPAN/
Delaware	http://ftp.lug.udel.edu/pub/CPAN ftp://ftp.lug.udel.edu/pub/CPAN
District of Columbia	ftp://ftp.dc.aleron.net/pub/CPAN/
Florida	ftp://ftp.cise.ufl.edu/pub/mirrors/CPAN/ http://mirror.csit.fsu.edu/pub/CPAN/ ftp://mirror.csit.fsu.edu/pub/CPAN/ http://cpan.mirrors.nks.net/
Illinois	http://uiarchive.uiuc.edu/mirrors/ftp/cpan.cse.msu.edu/ ftp://uiarchive.uiuc.edu/mirrors/ftp/cpan.cse.msu.edu/
Indiana	ftp://ftp.uwsg.iu.edu/pub/perl/CPAN/ http://cpan.netnitco.net/ ftp://cpan.netnitco.net/pub/mirrors/CPAN/ http://archive.progeny.com/CPAN/ ftp://archive.progeny.com/CPAN/ ftp://cpan.in-span.net/ http://csociety-ftp.ecn.purdue.edu/pub/CPAN ftp://csociety-ftp.ecn.purdue.edu/pub/CPAN
Kentucky	http://cpan.uky.edu/ ftp://cpan.uky.edu/pub/CPAN/
Massachusetts	ftp://ftp.ccs.neu.edu/net/mirrors/ftp.funet.fi/pub/language http://cpan.mirrors.netnumina.com/ ftp://mirrors.netnumina.com/cpan/
Michigan	ftp://cpan.cse.msu.edu/
New Jersey	ftp://ftp.cpanel.net/pub/CPAN/ http://cpan.teleglobe.net/ ftp://cpan.teleglobe.net/pub/CPAN
New York	ftp://ftp.exorbit.org/pub/perl/CPAN http://cpan.belfry.net/ http://cpan.thepirtgroup.com/ ftp://cpan.thepirtgroup.com/ ftp://ftp.stealth.net/pub/CPAN/

`http://www.rge.com/pub/languages/perl/`
`ftp://ftp.rge.com/pub/languages/perl/`
`ftp://mirrors.cloud9.net/pub/mirrors/CPAN/`

North Carolina

`ftp://ftp.duke.edu/pub/perl/`

Ohio

`ftp://ftp.loaded.net/pub/CPAN/`

Oklahoma

`ftp://ftp.ou.edu/mirrors/CPAN/`

Oregon

`ftp://ftp.orst.edu/pub/CPAN`

Pennsylvania

`http://ftp.epix.net/CPAN/`
`ftp://ftp.epix.net/pub/languages/perl/`
`http://mirrors.phenominet.com/pub/CPAN/`
`ftp://mirrors.phenominet.com/pub/CPAN/`
`http://cpan.pair.com/`
`ftp://cpan.pair.com/pub/CPAN/`
`ftp://carroll.cac.psu.edu/pub/CPAN/`

Tennessee

`ftp://ftp.sunsite.utk.edu/pub/CPAN/`

Texas

`http://ftp.sedl.org/pub/mirrors/CPAN/`
`ftp://mirror.telentente.com/pub/CPAN`

Utah

`ftp://mirror.xmission.com/CPAN/`

Virginia

`http://mirrors.rcn.net/pub/lang/CPAN/`
`ftp://mirrors.rcn.net/pub/lang/CPAN/`
`http://perl.secsup.org/`
`ftp://perl.secsup.org/pub/perl/`
`http://mirrors.phihost.com/CPAN/`
`ftp://mirrors.phihost.com/CPAN/`
`ftp://ruff.cs.jmu.edu/pub/CPAN/`
`http://perl.Liquidation.com/CPAN/`

ashington

`http://cpan.llarian.net/`
`ftp://cpan.llarian.net/pub/CPAN/`
`http://cpan.mirrorcentral.com/`
`ftp://ftp.mirrorcentral.com/pub/CPAN/`
`ftp://ftp-mirror.internap.com/pub/CPAN/`

Wisconsin

`http://mirror.sit.wisc.edu/pub/CPAN/`
`ftp://mirror.sit.wisc.edu/pub/CPAN/`

Oceania

Australia

```
http://ftp.planetmirror.com/pub/CPAN/  
ftp://ftp.planetmirror.com/pub/CPAN/  
ftp://mirror.aarnet.edu.au/pub/perl/CPAN/  
ftp://cpan.topend.com.au/pub/CPAN/
```

New Zealand

```
ftp://ftp.auckland.ac.nz/pub/perl/CPAN/  
http://cpan.soa.co.nz/CPAN/
```

South America

Argentina

```
ftp://mirrors.bannerlandia.com.ar/mirrors/CPAN/  
http://ftp.fcaglp.unlp.edu.ar/pub/CPAN/  
ftp://ftp.fcaglp.unlp.edu.ar/pub/CPAN/
```

Brazil

```
ftp://cpan.pop-mg.com.br/pub/CPAN/  
ftp://ftp.matrix.com.br/pub/perl/CPAN/
```

Chile

```
http://cpan.netglobalis.net/  
ftp://cpan.netglobalis.net/pub/CPAN/
```

RSYNC Mirrors

```
ftp.fcaglp.unlp.edu.ar::CPAN  
cpan.mirror.smartworker.org::CPAN  
theoryx5.uwinnipeg.ca::CPAN  
ftp.shellhung.org::CPAN  
ftp.funet.fi::CPAN  
ftp.u-paris10.fr::CPAN  
mir1.ovh.net::CPAN  
ftp.gwdg.de::FTP/languages/perl/CPAN/  
ftp.leo.org::CPAN  
CPAN.piksi.itb.ac.id::CPAN  
ftp.cbn.net.id::CPAN  
ftp.iglu.org.il::CPAN  
gusp.dyndns.org::cpan  
ftp.kddlabs.co.jp::cpan  
ftp.ayamura.org::pub/CPAN/  
mirror.averse.net::cpan  
cpan.teleglobe.net::CPAN  
ftp.sedl.org::cpan  
archive.progeny.com::CPAN  
cpan.teleglobe.net::CPAN  
ftp.lug.udel.edu::cpan  
mirrors.kernel.org::mirrors/CPAN  
mirrors.phenominet.com::CPAN  
mirror.csit.fsu.edu::CPAN  
csociety-ftp.ecn.purdue.edu::CPAN
```

For an up-to-date listing of CPAN sites, see <http://www.cpan.org/SITES> or <ftp://www.cpan.org/SITES>.

Modules: Creation, Use, and Abuse

(The following section is borrowed directly from Tim Bunce's modules file, available at your nearest CPAN site.)

Perl implements a class using a package, but the presence of a package doesn't imply the presence of a class. A package is just a namespace. A class is a package that provides subroutines that can be used as methods. A method is just a subroutine that expects, as its first argument, either the name of a package (for "static" methods), or a reference to something (for "virtual" methods).

A module is a file that (by convention) provides a class of the same name (sans the .pm), plus an import method in that class that can be called to fetch exported symbols. This module may implement some of its methods by loading dynamic C or C++ objects, but that should be totally transparent to the user of the module. Likewise, the module might set up an AUTOLOAD function to slurp in subroutine definitions on demand, but this is also transparent. Only the .pm file is required to exist. See [perlsb](#), [perltoot](#), and [AutoLoader](#) for details about the AUTOLOAD mechanism.

Guidelines for Module Creation

- Do similar modules already exist in some form?

If so, please try to reuse the existing modules either in whole or by inheriting useful features into a new class. If this is not practical try to get together with the module authors to work on extending or enhancing the functionality of the existing modules. A perfect example is the plethora of packages in perl4 for dealing with command line options.

If you are writing a module to expand an already existing set of modules, please coordinate with the author of the package. It helps if you follow the same naming scheme and module interaction scheme as the original author.

- Try to design the new module to be easy to extend and reuse.

Try to use `warnings;` (or use `warnings qw(...);`). Remember that you can add `no warnings qw(...);` to individual blocks of code that need less warnings.

Use blessed references. Use the two argument form of `bless` to bless into the class name given as the first parameter of the constructor, e.g.,:

```
sub new {
    my $class = shift;
    return bless {}, $class;
}
```

or even this if you'd like it to be used as either a static or a virtual method.

```
sub new {
    my $self = shift;
    my $class = ref($self) || $self;
    return bless {}, $class;
}
```

Pass arrays as references so more parameters can be added later (it's also faster). Convert functions into methods where appropriate. Split large methods into smaller more flexible ones. Inherit methods from other modules if appropriate.

Avoid class name tests like: `die "Invalid" unless ref $ref eq 'FOO'`. Generally you can delete the `eq 'FOO'` part with no harm at all. Let the objects look after themselves! Generally, avoid hard-wired class names as far as possible.

Avoid `< $r-Class::func()` where using `@ISA=qw(... Class ...)` and `< $r-func()` would work (see [perlbot](#) for more details).

Use `autosplit` so little used or newly added functions won't be a burden to programs that don't use them. Add test functions to the module after `__END__` either using `AutoSplit` or by saying:

```
eval join(' ', <main::DATA>) || die $@ unless caller();
```

Does your module pass the 'empty subclass' test? If you say `@SUBCLASS::ISA = qw(YOURCLASS);` your applications should be able to use `SUBCLASS` in exactly the same way as `YOURCLASS`. For example, does your application still work if you change: `$obj = new YOURCLASS;` into: `$obj = new SUBCLASS;`?

Avoid keeping any state information in your packages. It makes it difficult for multiple other packages to use yours. Keep state information in objects.

Always use `-w`.

Try to use `strict`; (or use `strict qw(...);`). Remember that you can add `no strict qw(...);` to individual blocks of code that need less strictness.

Always use `-w`.

Follow the guidelines in the `perlstyle(1)` manual.

Always use `-w`.

- Some simple style guidelines

The `perlstyle` manual supplied with Perl has many helpful points.

Coding style is a matter of personal taste. Many people evolve their style over several years as they learn what helps them write and maintain good code. Here's one set of assorted suggestions that seem to be widely used by experienced developers:

Use underscores to separate words. It is generally easier to read `$var_names_like_this` than `$VarNamesLikeThis`, especially for non-native speakers of English. It's also a simple rule that works consistently with `VAR_NAMES_LIKE_THIS`.

Package/Module names are an exception to this rule. Perl informally reserves lowercase module names for 'pragma' modules like `integer` and `strict`. Other modules normally begin with a capital letter and use mixed case with no underscores (need to be short and portable).

You may find it helpful to use letter case to indicate the scope or nature of a variable. For example:

```
$ALL_CAPS_HERE    constants only (beware clashes with Perl vars)
$Some_Caps_Here   package-wide global/static
$no_caps_here      function scope my() or local() variables
```

Function and method names seem to work best as all lowercase. e.g., `< $obj->as_string()`.

You can use a leading underscore to indicate that a variable or function should not be used outside the package that defined it.

- Select what to export.

Do NOT export method names!

Do NOT export anything else by default without a good reason!

Exports pollute the namespace of the module user. If you must export try to use `@EXPORT_OK` in preference to `@EXPORT` and avoid short or common names to reduce the risk of name clashes.

Generally anything not exported is still accessible from outside the module using the `ModuleName::item_name` (or `< $blessed_ref->method`) syntax. By convention you can use a leading underscore on names to indicate informally that they are 'internal' and not for public use.

(It is actually possible to get private functions by saying: `my $subref = sub { ... }; &$subref;` But there's no way to call that directly as a method, because a method must have a

name in the symbol table.)

As a general rule, if the module is trying to be object oriented then export nothing. If it's just a collection of functions then `@EXPORT_OK` anything but use `@EXPORT` with caution.

- Select a name for the module.

This name should be as descriptive, accurate, and complete as possible. Avoid any risk of ambiguity. Always try to use two or more whole words. Generally the name should reflect what is special about what the module does rather than how it does it. Please use nested module names to group informally or categorize a module. There should be a very good reason for a module not to have a nested name. Module names should begin with a capital letter.

Having 57 modules all called `Sort` will not make life easy for anyone (though having 23 called `Sort::Quick` is only marginally better :-). Imagine someone trying to install your module alongside many others. If in any doubt ask for suggestions in `comp.lang.perl.misc`.

If you are developing a suite of related modules/classes it's good practice to use nested classes with a common prefix as this will avoid namespace clashes. For example: `Xyz::Control`, `Xyz::View`, `Xyz::Model` etc. Use the modules in this list as a naming guide.

If adding a new module to a set, follow the original author's standards for naming modules and the interface to methods in those modules.

If developing modules for private internal or project specific use, that will never be released to the public, then you should ensure that their names will not clash with any future public module. You can do this either by using the reserved `Local::*` category or by using a category name that includes an underscore like `Foo_Corp::*`.

To be portable each component of a module name should be limited to 11 characters. If it might be used on MS-DOS then try to ensure each is unique in the first 8 characters. Nested modules make this easier.

- Have you got it right?

How do you know that you've made the right decisions? Have you picked an interface design that will cause problems later? Have you picked the most appropriate name? Do you have any questions?

The best way to know for sure, and pick up many helpful suggestions, is to ask someone who knows. `Comp.lang.perl.misc` is read by just about all the people who develop modules and it's the best place to ask.

All you need to do is post a short summary of the module, its purpose and interfaces. A few lines on each of the main methods is probably enough. (If you post the whole module it might be ignored by busy people – generally the very people you want to read it!)

Don't worry about posting if you can't say when the module will be ready – just say so in the message. It might be worth inviting others to help you, they may be able to complete it for you!

- README and other Additional Files.

It's well known that software developers usually fully document the software they write. If, however, the world is in urgent need of your software and there is not enough time to write the full documentation please at least provide a README file containing:

- A description of the module/package/extension etc.
- A copyright notice – see below.
- Prerequisites – what else you may need to have.
- How to build it – possible changes to `Makefile.PL` etc.

- How to install it.
- Recent changes in this release, especially incompatibilities
- Changes / enhancements you plan to make in the future.

If the README file seems to be getting too large you may wish to split out some of the sections into separate files: INSTALL, Copying, ToDo etc.

- Adding a Copyright Notice.

How you choose to license your work is a personal decision. The general mechanism is to assert your Copyright and then make a declaration of how others may copy/use/modify your work.

Perl, for example, is supplied with two types of licence: The GNU GPL and The Artistic Licence (see the files README, Copying, and Artistic). Larry has good reasons for NOT just using the GNU GPL.

My personal recommendation, out of respect for Larry, Perl, and the Perl community at large is to state something simply like:

```
Copyright (c) 1995 Your Name. All rights reserved.  
This program is free software; you can redistribute it and/or  
modify it under the same terms as Perl itself.
```

This statement should at least appear in the README file. You may also wish to include it in a Copying file and your source files. Remember to include the other words in addition to the Copyright.

- Give the module a version/issue/release number.

To be fully compatible with the Exporter and MakeMaker modules you should store your module's version number in a non-my package variable called \$VERSION. This should be a floating point number with at least two digits after the decimal (i.e., hundredths, e.g. \$VERSION = "0.01"). Don't use a "1.3.2" style version. See [Exporter](#) for details.

It may be handy to add a function or method to retrieve the number. Use the number in announcements and archive file names when releasing the module (ModuleName-1.02.tar.Z). See `perldoc ExtUtils::MakeMaker.pm` for details.

- How to release and distribute a module.

It's good idea to post an announcement of the availability of your module (or the module itself if small) to the `comp.lang.perl.announce` Usenet newsgroup. This will at least ensure very wide once-off distribution.

If possible, register the module with CPAN. You should include details of its location in your announcement.

Some notes about ftp archives: Please use a long descriptive file name that includes the version number. Most incoming directories will not be readable/listable, i.e., you won't be able to see your file after uploading it. Remember to send your email notification message as soon as possible after uploading else your file may get deleted automatically. Allow time for the file to be processed and/or check the file has been processed before announcing its location.

FTP Archives for Perl Modules:

Follow the instructions and links on:

```
http://www.cpan.org/modules/00modlist.long.html  
http://www.cpan.org/modules/04pause.html
```

or upload to one of these sites:

<https://pause.kbx.de/pause/>
<http://pause.perl.org/pause/>

and notify <modules@perl.org.

By using the WWW interface you can ask the Upload Server to mirror your modules from your ftp or WWW site into your own directory on CPAN!

Please remember to send me an updated entry for the Module list!

- Take care when changing a released module.

Always strive to remain compatible with previous released versions. Otherwise try to add a mechanism to revert to the old behavior if people rely on it. Document incompatible changes.

Guidelines for Converting Perl 4 Library Scripts into Modules

- There is no requirement to convert anything.

If it ain't broke, don't fix it! Perl 4 library scripts should continue to work with no problems. You may need to make some minor changes (like escaping non-array @'s in double quoted strings) but there is no need to convert a .pl file into a Module for just that.

- Consider the implications.

All Perl applications that make use of the script will need to be changed (slightly) if the script is converted into a module. Is it worth it unless you plan to make other changes at the same time?

- Make the most of the opportunity.

If you are going to convert the script to a module you can use the opportunity to redesign the interface. The guidelines for module creation above include many of the issues you should consider.

- The pl2pm utility will get you started.

This utility will read *.pl files (given as parameters) and write corresponding *.pm files. The pl2pm utilities does the following:

- Adds the standard Module prologue lines
- Converts package specifiers from ' to ::
- Converts die(...) to croak(...)
- Several other minor changes

Being a mechanical process pl2pm is not bullet proof. The converted code will need careful checking, especially any package statements. Don't delete the original .pl file till the new .pm one works!

Guidelines for Reusing Application Code

- Complete applications rarely belong in the Perl Module Library.
- Many applications contain some Perl code that could be reused.

Help save the world! Share your code in a form that makes it easy to reuse.

- Break-out the reusable code into one or more separate module files.
- Take the opportunity to reconsider and redesign the interfaces.
- In some cases the 'application' can then be reduced to a small

fragment of code built on top of the reusable modules. In these cases the application could invoked as:

```
% perl -e 'use Module::Name; method(@ARGV)' ...  
or  
% perl -mModule::Name ... (in perl5.002 or higher)
```

NOTE

Perl does not enforce private and public parts of its modules as you may have been used to in other languages like C++, Ada, or Modula-17. Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun.

The module and its user have a contract, part of which is common law, and part of which is "written". Part of the common law contract is that a module doesn't pollute any namespace it wasn't asked to. The written contract for the module (A.K.A. documentation) may make other provisions. But then you know when you use `RedefineTheWorld` that you're redefining the world and willing to take the consequences.

NAME

perlmodstyle – Perl module style guide

INTRODUCTION

This document attempts to describe the Perl Community's "best practice" for writing Perl modules. It extends the recommendations found in [perlstyle](#), which should be considered required reading before reading this document.

While this document is intended to be useful to all module authors, it is particularly aimed at authors who wish to publish their modules on CPAN.

The focus is on elements of style which are visible to the users of a module, rather than those parts which are only seen by the module's developers. However, many of the guidelines presented in this document can be extrapolated and applied successfully to a module's internals.

This document differs from [perlnewmod](#) in that it is a style guide rather than a tutorial on creating CPAN modules. It provides a checklist against which modules can be compared to determine whether they conform to best practice, without necessarily describing in detail how to achieve this.

All the advice contained in this document has been gleaned from extensive conversations with experienced CPAN authors and users. Every piece of advice given here is the result of previous mistakes. This information is here to help you avoid the same mistakes and the extra work that would inevitably be required to fix them.

The first section of this document provides an itemized checklist; subsequent sections provide a more detailed discussion of the items on the list. The final section, "Common Pitfalls", describes some of the most popular mistakes made by CPAN authors.

QUICK CHECKLIST

For more detail on each item in this checklist, see below.

Before you start

- Don't re-invent the wheel
- Patch, extend or subclass an existing module where possible
- Do one thing and do it well
- Choose an appropriate name

The API

- API should be understandable by the average programmer
- Simple methods for simple tasks
- Separate functionality from output
- Consistent naming of subroutines or methods
- Use named parameters (a hash or hashref) when there are more than two parameters

Stability

- Ensure your module works under `use strict` and `-w`
- Stable modules should maintain backwards compatibility

Documentation

- Write documentation in POD

- Document purpose, scope and target applications
- Document each publically accessible method or subroutine, including params and return values
- Give examples of use in your documentation
- Provide a README file and perhaps also release notes, changelog, etc
- Provide links to further information (URL, email)

Release considerations

- Specify pre-requisites in Makefile.PL
- Specify Perl version requirements with `use`
- Include tests with your module
- Choose a sensible and consistent version numbering scheme (X.YY is the common Perl module numbering scheme)
- Increment the version number for every change, no matter how small
- Package the module using "make dist"
- Choose an appropriate license (GPL/Artistic is a good default)

BEFORE YOU START WRITING A MODULE

Try not to launch headlong into developing your module without spending some time thinking first. A little forethought may save you a vast amount of effort later on.

Has it been done before?

You may not even need to write the module. Check whether it's already been done in Perl, and avoid re-inventing the wheel unless you have a good reason.

Good places to look for pre-existing modules include <http://search.cpan.org/> and asking on modules@perl.org

If an existing module **almost** does what you want, consider writing a patch, writing a subclass, or otherwise extending the existing module rather than rewriting it.

Do one thing and do it well

At the risk of stating the obvious, modules are intended to be modular. A Perl developer should be able to use modules to put together the building blocks of their application. However, it's important that the blocks are the right shape, and that the developer shouldn't have to use a big block when all they need is a small one.

Your module should have a clearly defined scope which is no longer than a single sentence. Can your module be broken down into a family of related modules?

Bad example:

"FooBar.pm provides an implementation of the FOO protocol and the related BAR standard."

Good example:

"Foo.pm provides an implementation of the FOO protocol. Bar.pm implements the related BAR protocol."

This means that if a developer only needs a module for the BAR standard, they should not be forced to install libraries for FOO as well.

What's in a name?

Make sure you choose an appropriate name for your module early on. This will help people find and remember your module, and make programming with your module more intuitive.

When naming your module, consider the following:

- Be descriptive (i.e. accurately describes the purpose of the module).
- Be consistent with existing modules.
- Reflect the functionality of the module, not the implementation.
- Avoid starting a new top-level hierarchy, especially if a suitable hierarchy already exists under which you could place your module.

You should contact modules@perl.org to ask them about your module name before publishing your module. You should also try to ask people who are already familiar with the module's application domain and the CPAN naming system. Authors of similar modules, or modules with similar names, may be a good place to start.

DESIGNING AND WRITING YOUR MODULE

Considerations for module design and coding:

To OO or not to OO?

Your module may be object oriented (OO) or not, or it may have both kinds of interfaces available. There are pros and cons of each technique, which should be considered when you design your API.

According to Damian Conway, you should consider using OO:

- When the system is large or likely to become so
- When the data is aggregated in obvious structures that will become objects
- When the types of data form a natural hierarchy that can make use of inheritance
- When operations on data vary according to data type (making polymorphic invocation of methods feasible)
- When it is likely that new data types may be later introduced into the system, and will need to be handled by existing code
- When interactions between data are best represented by overloaded operators
- When the implementation of system components is likely to change over time (and hence should be encapsulated)
- When the system design is itself object-oriented
- When large amounts of client code will use the software (and should be insulated from changes in its implementation)
- When many separate operations will need to be applied to the same set of data

Think carefully about whether OO is appropriate for your module. Gratuitous object orientation results in complex APIs which are difficult for the average module user to understand or use.

Designing your API

Your interfaces should be understandable by an average Perl programmer. The following guidelines may help you judge whether your API is sufficiently straightforward:

Write simple routines to do simple things.

It's better to have numerous simple routines than a few monolithic ones. If your routine changes its behaviour significantly based on its arguments, it's a sign that you should have two (or more) separate routines.

Separate functionality from output.

Return your results in the most generic form possible and allow the user to choose how to use them. The most generic form possible is usually a Perl data structure which can then be used to generate a text report, HTML, XML, a database query, or whatever else your users require.

If your routine iterates through some kind of list (such as a list of files, or records in a database) you may consider providing a callback so that users can manipulate each element of the list in turn. `File::Find` provides an example of this with its `find(\&wanted, $dir)` syntax.

Provide sensible shortcuts and defaults.

Don't require every module user to jump through the same hoops to achieve a simple result. You can always include optional parameters or routines for more complex or non-standard behaviour. If most of your users have to type a few almost identical lines of code when they start using your module, it's a sign that you should have made that behaviour a default. Another good indicator that you should use defaults is if most of your users call your routines with the same arguments.

Naming conventions

Your naming should be consistent. For instance, it's better to have:

```
display_day();
display_week();
display_year();
```

than

```
display_day();
week_display();
show_year();
```

This applies equally to method names, parameter names, and anything else which is visible to the user (and most things that aren't!)

Parameter passing

Use named parameters. It's easier to use a hash like this:

```
$obj->do_something(
    name => "wibble",
    type => "text",
    size => 1024,
);
```

... than to have a long list of unnamed parameters like this:

```
$obj->do_something("wibble", "text", 1024);
```

While the list of arguments might work fine for one, two or even three arguments, any more arguments become hard for the module user to remember, and hard for the module author to manage. If you want to add a new parameter you will have to add it to the end of the list for backward compatibility, and this will probably make your list order unintuitive. Also, if many elements may be undefined you may see the following unattractive method calls:

```
$obj->do_something(undef, undef, undef, undef, undef, undef, 1024);
```

Provide sensible defaults for parameters which have them. Don't make your users specify parameters which will almost always be the same.

The issue of whether to pass the arguments in a hash or a hashref is largely a matter of personal style.

The use of hash keys starting with a hyphen (`-name`) or entirely in upper case (`NAME`) is a relic of older versions of Perl in which ordinary lower case strings were not handled correctly by the `=>` operator. While some modules retain uppercase or hyphenated argument keys for historical reasons or as a matter of personal style, most new modules should use simple lower case keys. Whatever you choose, be consistent!

Strictness and warnings

Your module should run successfully under the strict pragma and should run without generating any warnings. Your module should also handle taint-checking where appropriate, though this can cause difficulties in many cases.

Backwards compatibility

Modules which are "stable" should not break backwards compatibility without at least a long transition phase and a major change in version number.

Error handling and messages

When your module encounters an error it should do one or more of:

- Return an undefined value.
- set `$Module::errstr` or similar (`errstr` is a common name used by DBI and other popular modules; if you choose something else, be sure to document it clearly).
- `warn()` or `carp()` a message to `STDERR`.
- `croak()` only when your module absolutely cannot figure out what to do. (`croak()` is a better version of `die()` for use within modules, which reports its errors from the perspective of the caller. See *Carp* for details of `croak()`, `carp()` and other useful routines.)
- As an alternative to the above, you may prefer to throw exceptions using the Error module.

Configurable error handling can be very useful to your users. Consider offering a choice of levels for warning and debug messages, an option to send messages to a separate file, a way to specify an error-handling routine, or other such features. Be sure to default all these options to the commonest use.

DOCUMENTING YOUR MODULE

POD

Your module should include documentation aimed at Perl developers. You should use Perl's "plain old documentation" (POD) for your general technical documentation, though you may wish to write additional documentation (white papers, tutorials, etc) in some other format. You need to cover the following subjects:

- A synopsis of the common uses of the module
- The purpose, scope and target applications of your module
- Use of each publically accessible method or subroutine, including parameters and return values
- Examples of use
- Sources of further information
- A contact email address for the author/maintainer

The level of detail in Perl module documentation generally goes from less detailed to more detailed. Your SYNOPSIS section should contain a minimal example of use (perhaps as little as one line of code; skip the unusual use cases or anything not needed by most users); the DESCRIPTION should describe your module in broad terms, generally in just a few paragraphs; more detail of the module's routines or methods, lengthy code examples, or other in-depth material should be given in subsequent sections.

Ideally, someone who's slightly familiar with your module should be able to refresh their memory without hitting "page down". As your reader continues through the document, they should receive a progressively greater amount of knowledge.

The recommended order of sections in Perl module documentation is:

- NAME
- SYNOPSIS
- DESCRIPTION
- One or more sections or subsections giving greater detail of available methods and routines and any other relevant information.
- BUGS/CAVEATS/etc
- AUTHOR
- SEE ALSO
- COPYRIGHT and LICENSE

Keep your documentation near the code it documents ("inline" documentation). Include POD for a given method right above that method's subroutine. This makes it easier to keep the documentation up to date, and avoids having to document each piece of code twice (once in POD and once in comments).

README, INSTALL, release notes, changelogs

Your module should also include a README file describing the module and giving pointers to further information (website, author email).

An INSTALL file should be included, and should contain simple installation instructions (usually "perl Makefile.PL; make; make install").

Release notes or changelogs should be produced for each release of your software describing user-visible changes to your module, in terms relevant to the user.

RELEASE CONSIDERATIONS

Version numbering

Version numbers should indicate at least major and minor releases, and possibly sub-minor releases. A major release is one in which most of the functionality has changed, or in which major new functionality is added. A minor release is one in which a small amount of functionality has been added or changed. Sub-minor version numbers are usually used for changes which do not affect functionality, such as documentation patches.

The most common CPAN version numbering scheme looks like this:

```
1.00, 1.10, 1.11, 1.20, 1.30, 1.31, 1.32
```

A correct CPAN version number is a floating point number with at least 2 digits after the decimal. You can test whether it conforms to CPAN by using

```
perl -MExtUtils::MakeMaker -le 'print MM->parse_version(shift)' 'Foo.pm'
```

If you want to release a 'beta' or 'alpha' version of a module but don't want CPAN.pm to list it as most recent use an '_' after the regular version number followed by at least 2 digits, eg. 1.20_01. If you do this, the following idiom is recommended:

```
$VERSION = "1.12_01";  
$XS_VERSION = $VERSION; # only needed if you have XS code  
$VERSION = eval $VERSION;
```

With that trick MakeMaker will only read the first line and thus read the underscore, while the perl interpreter will evaluate the \$VERSION and convert the string into a number. Later operations that treat \$VERSION as a number will then be able to do so without provoking a warning about \$VERSION not being a number.

Never release anything (even a one-word documentation patch) without incrementing the number. Even a one-word documentation patch should result in a change in version at the sub-minor level.

Pre-requisites

Module authors should carefully consider whether to rely on other modules, and which modules to rely on.

Most importantly, choose modules which are as stable as possible. In order of preference:

- Core Perl modules
- Stable CPAN modules
- Unstable CPAN modules
- Modules not available from CPAN

Specify version requirements for other Perl modules in the pre-requisites in your Makefile.PL.

Be sure to specify Perl version requirements both in Makefile.PL and with `require 5.6.1` or similar.

Testing

All modules should be tested before distribution (using "make disttest", and the tests should also be available to people installing the modules (using "make test").

The importance of these tests is proportional to the alleged stability of a module — a module which purports to be stable or which hopes to achieve wide use should adhere to as strict a testing regime as possible.

Useful modules to help you write tests (with minimum impact on your development process or your time) include `Test::Simple`, `Carp::Assert` and `Test::Inline`.

Packaging

Modules should be packaged using the standard MakeMaker tools, allowing them to be installed in a consistent manner. Use "make dist" to create your package.

Tools exist to help you build your module in a MakeMaker-friendly style. These include `ExtUtils::ModuleMaker` and `h2xs`. See also [perlnewmod](#).

Licensing

Make sure that your module has a license, and that the full text of it is included in the distribution (unless it's a common one and the terms of the license don't require you to include it).

If you don't know what license to use, dual licensing under the GPL and Artistic licenses (the same as Perl itself) is a good idea.

COMMON PITFALLS

Reinventing the wheel

There are certain application spaces which are already very, very well served by CPAN. One example is templating systems, another is date and time modules, and there are many more. While it is a rite of passage to write your own version of these things, please consider carefully whether the Perl world really needs you to publish it.

Trying to do too much

Your module will be part of a developer's toolkit. It will not, in itself, form the **entire** toolkit. It's tempting to add extra features until your code is a monolithic system rather than a set of modular building blocks.

Inappropriate documentation

Don't fall into the trap of writing for the wrong audience. Your primary audience is a reasonably experienced developer with at least a moderate understanding of your module's application domain, who's just downloaded your module and wants to start using it as quickly as possible.

Tutorials, end-user documentation, research papers, FAQs etc are not appropriate in a module's main documentation. If you really want to write these, include them as sub-documents such as `My::Module::Tutorial` or `My::Module::FAQ` and provide a link in the SEE ALSO section of the main documentation.

SEE ALSO

[*perlstyle*](#)

General Perl style guide

[*perlnewmod*](#)

How to create a new module

[*perlpod*](#)

POD documentation

[*podchecker*](#)

Verifies your POD's correctness

Testing tools

[*Test::Simple*](#), [*Test::Inline*](#), [*Carp::Assert*](#)

<http://pause.perl.org/>

Perl Authors Upload Server. Contains links to information for module authors.

Any good book on software engineering

AUTHOR

Kirrily "Skud" Robert <skud@cpan.org>

NAME

perlnewmod – preparing a new module for distribution

DESCRIPTION

This document gives you some suggestions about how to go about writing Perl modules, preparing them for distribution, and making them available via CPAN.

One of the things that makes Perl really powerful is the fact that Perl hackers tend to want to share the solutions to problems they've faced, so you and I don't have to battle with the same problem again.

The main way they do this is by abstracting the solution into a Perl module. If you don't know what one of these is, the rest of this document isn't going to be much use to you. You're also missing out on an awful lot of useful code; consider having a look at [perlmod](#), [perlmodlib](#) and [perlmodinstall](#) before coming back here.

When you've found that there isn't a module available for what you're trying to do, and you've had to write the code yourself, consider packaging up the solution into a module and uploading it to CPAN so that others can benefit.

Warning

We're going to primarily concentrate on Perl-only modules here, rather than XS modules. XS modules serve a rather different purpose, and you should consider different things before distributing them – the popularity of the library you are gluing, the portability to other operating systems, and so on. However, the notes on preparing the Perl side of the module and packaging and distributing it will apply equally well to an XS module as a pure-Perl one.

What should I make into a module?

You should make a module out of any code that you think is going to be useful to others. Anything that's likely to fill a hole in the communal library and which someone else can slot directly into their program. Any part of your code which you can isolate and extract and plug into something else is a likely candidate.

Let's take an example. Suppose you're reading in data from a local format into a hash-of-hashes in Perl, turning that into a tree, walking the tree and then piping each node to an Acme Transmogrifier Server.

Now, quite a few people have the Acme Transmogrifier, and you've had to write something to talk the protocol from scratch – you'd almost certainly want to make that into a module. The level at which you pitch it is up to you: you might want protocol-level modules analogous to [Net::SMTP/Net::SMTP](#) which then talk to higher level modules analogous to [Mail::Send/Mail::Send](#). The choice is yours, but you do want to get a module out for that server protocol.

Nobody else on the planet is going to talk your local data format, so we can ignore that. But what about the thing in the middle? Building tree structures from Perl variables and then traversing them is a nice, general problem, and if nobody's already written a module that does that, you might want to modularise that code too.

So hopefully you've now got a few ideas about what's good to modularise. Let's now see how it's done.

Step-by-step: Preparing the ground

Before we even start scraping out the code, there are a few things we'll want to do in advance.

Look around

Dig into a bunch of modules to see how they're written. I'd suggest starting with [Text::Tabs/Text::Tabs](#), since it's in the standard library and is nice and simple, and then looking at something like [Time::Zone/Time::Zone](#), [File::Copy/File::Copy](#) and then some of the `Mail::*` modules if you're planning on writing object oriented code.

These should give you an overall feel for how modules are laid out and written.

Check it's new

There are a lot of modules on CPAN, and it's easy to miss one that's similar to what you're planning on contributing. Have a good plough through the modules list and the *by-module* directories, and make

sure you're not the one reinventing the wheel!

Discuss the need

You might love it. You might feel that everyone else needs it. But there might not actually be any real demand for it out there. If you're unsure about the demand your module will have, consider sending out feelers on the `comp.lang.perl.modules` newsgroup, or as a last resort, ask the modules list at `modules@perl.org`. Remember that this is a closed list with a very long turn-around time – be prepared to wait a good while for a response from them.

Choose a name

Perl modules included on CPAN have a naming hierarchy you should try to fit in with. See [perlmodlib](#) for more details on how this works, and browse around CPAN and the modules list to get a feel of it. At the very least, remember this: modules should be title capitalised, (`This::Thing`) fit in with a category, and explain their purpose succinctly.

Check again

While you're doing that, make really sure you haven't missed a module similar to the one you're about to write.

When you've got your name sorted out and you're sure that your module is wanted and not currently available, it's time to start coding.

Step-by-step: Making the module

Start with [h2xs](#)

Originally a utility to convert C header files into XS modules, [h2xs/h2xs](#) has become a useful utility for churning out skeletons for Perl-only modules as well. If you don't want to use the [Autoloader/Autoloader](#) which splits up big modules into smaller subroutine-sized chunks, you'll say something like this:

```
h2xs -AX -n Net::Acme
```

The `-A` omits the Autoloader code, `-X` omits XS elements, and `-n` specifies the name of the module.

Use [strict/strict](#) and [warnings/warnings](#)

A module's code has to be warning and strict-clean, since you can't guarantee the conditions that it'll be used under. Besides, you wouldn't want to distribute code that wasn't warning or strict-clean anyway, right?

Use [Carp/Carp](#)

The [Carp/Carp](#) module allows you to present your error messages from the caller's perspective; this gives you a way to signal a problem with the caller and not your module. For instance, if you say this:

```
warn "No hostname given";
```

the user will see something like this:

```
No hostname given at /usr/local/lib/perl5/site_perl/5.6.0/Net/Acme.pm  
line 123.
```

which looks like your module is doing something wrong. Instead, you want to put the blame on the user, and say this:

```
No hostname given at bad_code, line 10.
```

You do this by using [Carp/Carp](#) and replacing your `warn`s with `carps`. If you need to die, say `croak` instead. However, keep `warn` and `die` in place for your sanity checks – where it really is your module at fault.

Use [Exporter/Exporter](#) – wisely!

`h2xs` provides stubs for [Exporter/Exporter](#), which gives you a standard way of exporting symbols and subroutines from your module into the caller's namespace. For instance, saying `use Net::Acme`

`qw(&frob)` would import the `frob` subroutine.

The package variable `@EXPORT` will determine which symbols will get exported when the caller simply says `use Net::Acme` – you will hardly ever want to put anything in there. `@EXPORT_OK`, on the other hand, specifies which symbols you’re willing to export. If you do want to export a bunch of symbols, use the `%EXPORT_TAGS` and define a standard export set – look at [Exporter](#) for more details.

Use [plain old documentation/perlpod](#)

The work isn’t over until the paperwork is done, and you’re going to need to put in some time writing some documentation for your module. `h2xs` will provide a stub for you to fill in; if you’re not sure about the format, look at [perlpod](#) for an introduction. Provide a good synopsis of how your module is used in code, a description, and then notes on the syntax and function of the individual subroutines or methods. Use Perl comments for developer notes and POD for end–user notes.

Write tests

You’re encouraged to create self–tests for your module to ensure it’s working as intended on the myriad platforms Perl supports; if you upload your module to CPAN, a host of testers will build your module and send you the results of the tests. Again, `h2xs` provides a test framework which you can extend – you should do something more than just checking your module will compile.

Write the README

If you’re uploading to CPAN, the automated gremlins will extract the README file and place that in your CPAN directory. It’ll also appear in the main *by–module* and *by–category* directories if you make it onto the modules list. It’s a good idea to put here what the module actually does in detail, and the user–visible changes since the last release.

Step–by–step: Distributing your module

Get a CPAN user ID

Every developer publishing modules on CPAN needs a CPAN ID. See the instructions at <http://www.cpan.org/modules/04pause.html> (or equivalent on your nearest mirror) to find out how to do this.

```
perl Makefile.PL; make test; make dist
```

Once again, `h2xs` has done all the work for you. It produces the standard `Makefile.PL` you’ll have seen when you downloaded and installs modules, and this produces a `Makefile` with a `dist` target.

Once you’ve ensured that your module passes its own tests – always a good thing to make sure – you can make `dist`, and the `Makefile` will hopefully produce you a nice tarball of your module, ready for upload.

Upload the tarball

The email you got when you received your CPAN ID will tell you how to log in to PAUSE, the Perl Authors Upload SErver. From the menus there, you can upload your module to CPAN.

Announce to the modules list

Once uploaded, it’ll sit unnoticed in your author directory. If you want it connected to the rest of the CPAN, you’ll need to tell the modules list about it. The best way to do this is to email them a line in the style of the modules list, like this:

```
Net::Acme  bdpOP      Interface to Acme Frobnicator servers      FOOBAR
^          ^^^^^^    ^
|          |         |
|          |         | Module description                    | Your ID
|          |         |
|          |         | \-Public Licence: (p)standard Perl, (g)GPL, (b)BSD,
|          |         |                    (l)LGPL, (a)rtistic, (o)ther
|          |         |
|          |         | \- Interface: (O)OP, (r)eferences, (h)ybrid, (f)unctions
|          |         |
```

```

|
| | \-- Language: (p)ure Perl, C(+)+, (h)ybrid, (C), (o)ther
| |
Module | \--- Support: (d)eveloper, (m)ailing list, (u)senet, (n)one
Name   |
       | \---- Development: (i)dea, (c)onstructions, (a)lpha, (b)eta,
       |                   (R)eleased, (M)ature, (S)tandard

```

plus a description of the module and why you think it should be included. If you hear nothing back, that means your module will probably appear on the modules list at the next update. Don't try subscribing to `modules@perl.org`; it's not another mailing list. Just have patience.

Announce to clpa

If you have a burning desire to tell the world about your release, post an announcement to the moderated `comp.lang.perl.announce` newsgroup.

Fix bugs!

Once you start accumulating users, they'll send you bug reports. If you're lucky, they'll even send you patches. Welcome to the joys of maintaining a software project...

AUTHOR

Simon Cozens, simon@cpan.org

SEE ALSO

[perlmod](#), [perlmodlib](#), [perlmodinstall](#), [h2xs](#), [strict](#), [Carp](#), [Exporter](#), [perlpod](#), [Test](#), [ExtUtils::MakeMaker](#), <http://www.cpan.org/>, Ken Williams' tutorial on building your own module at http://mathforum.org/~ken/perl_modules.html

NAME

perlnumber – semantics of numbers and numeric operations in Perl

SYNOPSIS

```
$n = 1234;           # decimal integer
$n = 0b1110011;      # binary integer
$n = 01234;          # octal integer
$n = 0x1234;          # hexadecimal integer
$n = 12.34e-56;       # exponential notation
$n = "-12.34e56";     # number specified as a string
$n = "1234";          # number specified as a string
$n = v49.50.51.52;    # number specified as a string, which in
                       # turn is specified in terms of numbers :-)
```

DESCRIPTION

This document describes how Perl internally handles numeric values.

Perl's operator overloading facility is completely ignored here. Operator overloading allows user-defined behaviors for numbers, such as operations over arbitrarily large integers, floating points numbers with arbitrary precision, operations over "exotic" numbers such as modular arithmetic or p-adic arithmetic, and so on. See [overload](#) for details.

Storing numbers

Perl can internally represent numbers in 3 different ways: as native integers, as native floating point numbers, and as decimal strings. Decimal strings may have an exponential notation part, as in "12.34e-56". *Native* here means "a format supported by the C compiler which was used to build perl".

The term "native" does not mean quite as much when we talk about native integers, as it does when native floating point numbers are involved. The only implication of the term "native" on integers is that the limits for the maximal and the minimal supported true integral quantities are close to powers of 2. However, "native" floats have a most fundamental restriction: they may represent only those numbers which have a relatively "short" representation when converted to a binary fraction. For example, 0.9 cannot be represented by a native float, since the binary fraction for 0.9 is infinite:

```
binary0.1110011001100...
```

with the sequence 1100 repeating again and again. In addition to this limitation, the exponent of the binary number is also restricted when it is represented as a floating point number. On typical hardware, floating point values can store numbers with up to 53 binary digits, and with binary exponents between -1024 and 1024. In decimal representation this is close to 16 decimal digits and decimal exponents in the range of -304..304. The upshot of all this is that Perl cannot store a number like 12345678901234567 as a floating point number on such architectures without loss of information.

Similarly, decimal strings can represent only those numbers which have a finite decimal expansion. Being strings, and thus of arbitrary length, there is no practical limit for the exponent or number of decimal digits for these numbers. (But realize that what we are discussing the rules for just the *storage* of these numbers. The fact that you can store such "large" numbers does not mean that the *operations* over these numbers will use all of the significant digits. See ["Numeric operators and numeric conversions"](#) for details.)

In fact numbers stored in the native integer format may be stored either in the signed native form, or in the unsigned native form. Thus the limits for Perl numbers stored as native integers would typically be $-2^{31}..2^{32}-1$, with appropriate modifications in the case of 64-bit integers. Again, this does not mean that Perl can do operations only over integers in this range: it is possible to store many more integers in floating point format.

Summing up, Perl numeric values can store only those numbers which have a finite decimal expansion or a "short" binary expansion.

Numeric operators and numeric conversions

As mentioned earlier, Perl can store a number in any one of three formats, but most operators typically understand only one of those formats. When a numeric value is passed as an argument to such an operator, it will be converted to the format understood by the operator.

Six such conversions are possible:

native integer	--> native floating point	(*)
native integer	--> decimal string	
native floating_point	--> native integer	(*)
native floating_point	--> decimal string	(*)
decimal string	--> native integer	
decimal string	--> native floating point	(*)

These conversions are governed by the following general rules:

- If the source number can be represented in the target form, that representation is used.
- If the source number is outside of the limits representable in the target form, a representation of the closest limit is used. (*Loss of information*)
- If the source number is between two numbers representable in the target form, a representation of one of these numbers is used. (*Loss of information*)
- In `< native floating point - native integer` conversions the magnitude of the result is less than or equal to the magnitude of the source. (*"Rounding to zero"*.)
- If the `< decimal string - native integer` conversion cannot be done without loss of information, the result is compatible with the conversion sequence `< decimal_string - native_floating_point - native_integer`. In particular, rounding is strongly biased to 0, though a number like `"0.99999999999999999999"` has a chance of being rounded to 1.

RESTRICTION: The conversions marked with (*) above involve steps performed by the C compiler. In particular, bugs/features of the compiler used may lead to breakage of some of the above rules.

Flavors of Perl numeric operations

Perl operations which take a numeric argument treat that argument in one of four different ways: they may force it to one of the integer/floating/ string formats, or they may behave differently depending on the format of the operand. Forcing a numeric value to a particular format does not change the number stored in the value.

All the operators which need an argument in the integer format treat the argument as in modular arithmetic, e.g., `mod 2**32` on a 32-bit architecture. `sprintf "%u", -1` therefore provides the same result as `sprintf "%u", ~0`.

Arithmetic operators

The binary operators `+` `-` `*` `/` `==` `!=` `>` `<` `>=` `<=` and the unary operators `- abs` and `-` will attempt to convert arguments to integers. If both conversions are possible without loss of precision, and the operation can be performed without loss of precision then the integer result is used. Otherwise arguments are converted to floating point format and the floating point result is used. The caching of conversions (as described above) means that the integer conversion does not throw away fractional parts on floating point numbers.

++ `++` behaves as the other operators above, except that if it is a string matching the format `/^[a-zA-Z]*[0-9]*\z/` the string increment described in [perlop](#) is used.

Arithmetic operators during `use integer`

In scopes where `use integer;` is in force, nearly all the operators listed above will force their argument(s) into integer format, and return an integer result. The exceptions, `abs`, `++` and `-`, do not change their behavior with `use integer;`

Other mathematical operators

Operators such as `**`, `sin` and `exp` force arguments to floating point format.

Bitwise operators

Arguments are forced into the integer format if not strings.

Bitwise operators during `use integer`

forces arguments to integer format. Also shift operations internally use signed integers rather than the default unsigned.

Operators which expect an integer

force the argument into the integer format. This is applicable to the third and fourth arguments of `sysread`, for example.

Operators which expect a string

force the argument into the string format. For example, this is applicable to `printf "%s", $value`.

Though forcing an argument into a particular form does not change the stored number, Perl remembers the result of such conversions. In particular, though the first such conversion may be time-consuming, repeated operations will not need to redo the conversion.

AUTHOR

Ilya Zakharevich ilya@math.ohio-state.edu

Editorial adjustments by Gurusamy Sarathy <gsar@ActiveState.com>

Updates for 5.8.0 by Nicholas Clark <nick@ccl4.org>

SEE ALSO

[overload](#), [perlop](#)

NAME

perlobj – Perl objects

DESCRIPTION

First you need to understand what references are in Perl. See [perlref](#) for that. Second, if you still find the following reference work too complicated, a tutorial on object-oriented programming in Perl can be found in [perltoot](#) and [perltooc](#).

If you're still with us, then here are three very simple definitions that you should find reassuring.

1. An object is simply a reference that happens to know which class it belongs to.
2. A class is simply a package that happens to provide methods to deal with object references.
3. A method is simply a subroutine that expects an object reference (or a package name, for class methods) as the first argument.

We'll cover these points now in more depth.

An Object is Simply a Reference

Unlike say C++, Perl doesn't provide any special syntax for constructors. A constructor is merely a subroutine that returns a reference to something "blessed" into a class, generally the class that the subroutine is defined in. Here is a typical constructor:

```
package Critter;
sub new { bless {} }
```

That word new isn't special. You could have written a construct this way, too:

```
package Critter;
sub spawn { bless {} }
```

This might even be preferable, because the C++ programmers won't be tricked into thinking that new works in Perl as it does in C++. It doesn't. We recommend that you name your constructors whatever makes sense in the context of the problem you're solving. For example, constructors in the Tk extension to Perl are named after the widgets they create.

One thing that's different about Perl constructors compared with those in C++ is that in Perl, they have to allocate their own memory. (The other thing is that they don't automatically call overridden base-class constructors.) The {} allocates an anonymous hash containing no key/value pairs, and returns it. The `bless()` takes that reference and tells the object it references that it's now a Critter, and returns the reference. This is for convenience, because the referenced object itself knows that it has been blessed, and the reference to it could have been returned directly, like this:

```
sub new {
    my $self = {};
    bless $self;
    return $self;
}
```

You often see such a thing in more complicated constructors that wish to call methods in the class as part of the construction:

```
sub new {
    my $self = {};
    bless $self;
    $self->initialize();
    return $self;
}
```

If you care about inheritance (and you should; see [Modules: Creation, Use, and Abuse in perlmodlib](#)), then

you want to use the two-arg form of `bless` so that your constructors may be inherited:

```
sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    $self->initialize();
    return $self;
}
```

Or if you expect people to call not just `< CLASS-new()` but also `< $obj-new()`, then use something like this. The `initialize()` method used will be of whatever `$class` we blessed the object into:

```
sub new {
    my $this = shift;
    my $class = ref($this) || $this;
    my $self = {};
    bless $self, $class;
    $self->initialize();
    return $self;
}
```

Within the class package, the methods will typically deal with the reference as an ordinary reference. Outside the class package, the reference is generally treated as an opaque value that may be accessed only through the class's methods.

Although a constructor can in theory re-bless a referenced object currently belonging to another class, this is almost certainly going to get you into trouble. The new class is responsible for all cleanup later. The previous blessing is forgotten, as an object may belong to only one class at a time. (Although of course it's free to inherit methods from many classes.) If you find yourself having to do this, the parent class is probably misbehaving, though.

A clarification: Perl objects are blessed. References are not. Objects know which package they belong to. References do not. The `bless()` function uses the reference to find the object. Consider the following example:

```
$a = {};
$b = $a;
bless $a, BLAH;
print "\$b is a ", ref($b), "\n";
```

This reports `$b` as being a `BLAH`, so obviously `bless()` operated on the object and not on the reference.

A Class is Simply a Package

Unlike say C++, Perl doesn't provide any special syntax for class definitions. You use a package as a class by putting method definitions into the class.

There is a special array within each package called `@ISA`, which says where else to look for a method if you can't find it in the current package. This is how Perl implements inheritance. Each element of the `@ISA` array is just the name of another package that happens to be a class package. The classes are searched (depth first) for missing methods in the order that they occur in `@ISA`. The classes accessible through `@ISA` are known as base classes of the current class.

All classes implicitly inherit from class `UNIVERSAL` as their last base class. Several commonly used methods are automatically supplied in the `UNIVERSAL` class; see *"Default UNIVERSAL methods"* for more details.

If a missing method is found in a base class, it is cached in the current class for efficiency. Changing `@ISA` or defining new subroutines invalidates the cache and causes Perl to do the lookup again.

If neither the current class, its named base classes, nor the UNIVERSAL class contains the requested method, these three places are searched all over again, this time looking for a method named AUTOLOAD(). If an AUTOLOAD is found, this method is called on behalf of the missing method, setting the package global \$AUTOLOAD to be the fully qualified name of the method that was intended to be called.

If none of that works, Perl finally gives up and complains.

If you want to stop the AUTOLOAD inheritance say simply

```
sub AUTOLOAD;
```

and the call will die using the name of the sub being called.

Perl classes do method inheritance only. Data inheritance is left up to the class itself. By and large, this is not a problem in Perl, because most classes model the attributes of their object using an anonymous hash, which serves as its own little namespace to be carved up by the various classes that might want to do something with the object. The only problem with this is that you can't sure that you aren't using a piece of the hash that isn't already used. A reasonable workaround is to prepend your fieldname in the hash with the package name.

```
sub bump {
    my $self = shift;
    $self->{ __PACKAGE__ . ".count" }++;
}
```

A Method is Simply a Subroutine

Unlike say C++, Perl doesn't provide any special syntax for method definition. (It does provide a little syntax for method invocation though. More on that later.) A method expects its first argument to be the object (reference) or package (string) it is being invoked on. There are two ways of calling methods, which we'll call class methods and instance methods.

A class method expects a class name as the first argument. It provides functionality for the class as a whole, not for any individual object belonging to the class. Constructors are often class methods, but see [perltoot](#) and [perltooc](#) for alternatives. Many class methods simply ignore their first argument, because they already know what package they're in and don't care what package they were invoked via. (These aren't necessarily the same, because class methods follow the inheritance tree just like ordinary instance methods.) Another typical use for class methods is to look up an object by name:

```
sub find {
    my ($class, $name) = @_;
    $objtable{$name};
}
```

An instance method expects an object reference as its first argument. Typically it shifts the first argument into a "self" or "this" variable, and then uses that as an ordinary reference.

```
sub display {
    my $self = shift;
    my @keys = @_ ? @_ : sort keys %$self;
    foreach $key (@keys) {
        print "\t$key => $self->{$key}\n";
    }
}
```

Method Invocation

For various historical and other reasons, Perl offers two equivalent ways to write a method call. The simpler and more common way is to use the arrow notation:

```
my $fred = Critter->find("Fred");
$fred->display("Height", "Weight");
```

You should already be familiar with the use of the `< -` operator with references. In fact, since `$fred` above is a reference to an object, you could think of the method call as just another form of dereferencing.

Whatever is on the left side of the arrow, whether a reference or a class name, is passed to the method subroutine as its first argument. So the above code is mostly equivalent to:

```
my $fred = Critter::find("Critter", "Fred");
Critter::display($fred, "Height", "Weight");
```

How does Perl know which package the subroutine is in? By looking at the left side of the arrow, which must be either a package name or a reference to an object, i.e. something that has been blessed to a package. Either way, that's the package where Perl starts looking. If that package has no subroutine with that name, Perl starts looking for it in any base classes of that package, and so on.

If you need to, you *can* force Perl to start looking in some other package:

```
my $barney = MyCritter->Critter::find("Barney");
$barney->Critter::display("Height", "Weight");
```

Here `MyCritter` is presumably a subclass of `Critter` that defines its own versions of `find()` and `display()`. We haven't specified what those methods do, but that doesn't matter above since we've forced Perl to start looking for the subroutines in `Critter`.

As a special case of the above, you may use the `SUPER` pseudo-class to tell Perl to start looking for the method in the packages named in the current class's `@ISA` list.

```
package MyCritter;
use base 'Critter';    # sets @MyCritter::ISA = ('Critter');

sub display {
    my ($self, @args) = @_;
    $self->SUPER::display("Name", @args);
}
```

Instead of a class name or an object reference, you can also use any expression that returns either of those on the left side of the arrow. So the following statement is valid:

```
Critter->find("Fred")->display("Height", "Weight");
```

and so is the following:

```
my $fred = (reverse "rettirC")->find(reverse "derF");
```

Indirect Object Syntax

The other way to invoke a method is by using the so-called "indirect object" notation. This syntax was available in Perl 4 long before objects were introduced, and is still used with filehandles like this:

```
print STDERR "help!!!\n";
```

The same syntax can be used to call either object or class methods.

```
my $fred = find Critter "Fred";
display $fred "Height", "Weight";
```

Notice that there is no comma between the object or class name and the parameters. This is how Perl can tell you want an indirect method call instead of an ordinary subroutine call.

But what if there are no arguments? In that case, Perl must guess what you want. Even worse, it must make that guess *at compile time*. Usually Perl gets it right, but when it doesn't you get a function call compiled as a method, or vice versa. This can introduce subtle bugs that are hard to detect.

For example, a call to a method `new` in indirect notation — as C++ programmers are wont to make — can be miscompiled into a subroutine call if there's already a `new` function in scope. You'd end up calling the current package's `new` as a subroutine, rather than the desired class's method. The compiler tries to cheat by

remembering bareword `requires`, but the grief when it messes up just isn't worth the years of debugging it will take you to track down such subtle bugs.

There is another problem with this syntax: the indirect object is limited to a name, a scalar variable, or a block, because it would have to do too much lookahead otherwise, just like any other postfix dereference in the language. (These are the same quirky rules as are used for the filehandle slot in functions like `print` and `printf`.) This can lead to horribly confusing precedence problems, as in these next two lines:

```
move $obj->{FIELD};           # probably wrong!
move $ary[$i];                # probably wrong!
```

Those actually parse as the very surprising:

```
$obj->move->{FIELD};           # Well, lookee here
$ary->move([ $i ]);            # Didn't expect this one, eh?
```

Rather than what you might have expected:

```
$obj->{FIELD}->move();         # You should be so lucky.
$ary[$i]->move;                # Yeah, sure.
```

To get the correct behavior with indirect object syntax, you would have to use a block around the indirect object:

```
move { $obj->{FIELD} };
move { $ary[$i] };
```

Even then, you still have the same potential problem if there happens to be a function named `move` in the current package. **The `< -` notation suffers from neither of these disturbing ambiguities, so we recommend you use it exclusively. However, you may still end up having to read code using the indirect object notation, so it's important to be familiar with it.**

Default UNIVERSAL methods

The `UNIVERSAL` package automatically contains the following methods that are inherited by all other classes:

`isa(CLASS)`

`isa` returns *true* if its object is blessed into a subclass of `CLASS`

You can also call `UNIVERSAL::isa` as a subroutine with two arguments. The first does not need to be an object or even a reference. This allows you to check what a reference points to, or whether something is a reference of a given type. Example

```
if (UNIVERSAL::isa($ref, 'ARRAY')) {
    #...
}
```

To determine if a reference is a blessed object, you can write

```
print "It's an object\n" if UNIVERSAL::isa($val, 'UNIVERSAL');
```

`can(METHOD)`

`can` checks to see if its object has a method called `METHOD`, if it does then a reference to the sub is returned, if it does not then *undef* is returned.

`UNIVERSAL::can` can also be called as a subroutine with two arguments. It'll always return *undef* if its first argument isn't an object or a class name. So here's another way to check if a reference is a blessed object

```
print "It's still an object\n" if UNIVERSAL::can($val, 'can');
```

You can also use the `blessed` function of `Scalar::Util`:

```
use Scalar::Util 'blessed';

my $blessing = blessed $suspected_object;
```

blessed returns the name of the package the argument has been blessed into, or undef.

VERSION([NEED])

VERSION returns the version number of the class (package). If the NEED argument is given then it will check that the current version (as defined by the \$VERSION variable in the given package) not less than NEED; it will die if this is not the case. This method is normally called as a class method. This method is called automatically by the VERSION form of use.

```
use A 1.2 qw(some imported subs);
# implies:
A->VERSION(1.2);
```

NOTE: can directly uses Perl's internal code for method lookup, and isa uses a very similar method and cache-ing strategy. This may cause strange effects if the Perl code dynamically changes @ISA in any package.

You may add other methods to the UNIVERSAL class via Perl or XS code. You do not need to use UNIVERSAL to make these methods available to your program (and you should not do so).

Destructors

When the last reference to an object goes away, the object is automatically destroyed. (This may even be after you exit, if you've stored references in global variables.) If you want to capture control just before the object is freed, you may define a DESTROY method in your class. It will automatically be called at the appropriate moment, and you can do any extra cleanup you need to do. Perl passes a reference to the object under destruction as the first (and only) argument. Beware that the reference is a read-only value, and cannot be modified by manipulating \$_[0] within the destructor. The object itself (i.e. the thingy the reference points to, namely \${ \$_[0] }, @{\$ \$_[0] }, %{\$ \$_[0] } etc.) is not similarly constrained.

If you arrange to re-bless the reference before the destructor returns, perl will again call the DESTROY method for the re-blessed object after the current one returns. This can be used for clean delegation of object destruction, or for ensuring that destructors in the base classes of your choosing get called. Explicitly calling DESTROY is also possible, but is usually never needed.

Do not confuse the previous discussion with how objects *CONTAINED* in the current one are destroyed. Such objects will be freed and destroyed automatically when the current object is freed, provided no other references to them exist elsewhere.

Summary

That's about all there is to it. Now you need just to go off and buy a book about object-oriented design methodology, and bang your forehead with it for the next six months or so.

Two-Phased Garbage Collection

For most purposes, Perl uses a fast and simple, reference-based garbage collection system. That means there's an extra dereference going on at some level, so if you haven't built your Perl executable using your C compiler's -O flag, performance will suffer. If you *have* built Perl with cc -O, then this probably won't matter.

A more serious concern is that unreachable memory with a non-zero reference count will not normally get freed. Therefore, this is a bad idea:

```
{
    my $a;
    $a = \ $a;
}
```

Even though \$a *should* go away, it can't. When building recursive data structures, you'll have to break the self-reference yourself explicitly if you don't care to leak. For example, here's a self-referential node such

as one might use in a sophisticated tree structure:

```
sub new_node {
    my $self = shift;
    my $class = ref($self) || $self;
    my $node = {};
    $node->{LEFT} = $node->{RIGHT} = $node;
    $node->{DATA} = [ @_ ];
    return bless $node => $class;
}
```

If you create nodes like that, they (currently) won't go away unless you break their self reference yourself. (In other words, this is not to be construed as a feature, and you shouldn't depend on it.)

Almost.

When an interpreter thread finally shuts down (usually when your program exits), then a rather costly but complete mark-and-sweep style of garbage collection is performed, and everything allocated by that thread gets destroyed. This is essential to support Perl as an embedded or a multithreadable language. For example, this program demonstrates Perl's two-phased garbage collection:

```
#!/usr/bin/perl
package Subtle;

sub new {
    my $test;
    $test = \ $test;
    warn "CREATING " . \ $test;
    return bless \ $test;
}

sub DESTROY {
    my $self = shift;
    warn "DESTROYING $self";
}

package main;

warn "starting program";
{
    my $a = Subtle->new;
    my $b = Subtle->new;
    $$a = 0; # break selfref
    warn "leaving block";
}

warn "just exited block";
warn "time to die...";
exit;
```

When run as */tmp/test*, the following output is produced:

```
starting program at /tmp/test line 18.
CREATING SCALAR(0x8e5b8) at /tmp/test line 7.
CREATING SCALAR(0x8e57c) at /tmp/test line 7.
leaving block at /tmp/test line 23.
DESTROYING Subtle=SCALAR(0x8e5b8) at /tmp/test line 13.
just exited block at /tmp/test line 26.
time to die... at /tmp/test line 27.
DESTROYING Subtle=SCALAR(0x8e57c) during global destruction.
```

Notice that "global destruction" bit there? That's the thread garbage collector reaching the unreachable.

Objects are always destructed, even when regular refs aren't. Objects are destructed in a separate pass before ordinary refs just to prevent object destructors from using refs that have been themselves destructed. Plain refs are only garbage-collected if the destruct level is greater than 0. You can test the higher levels of global destruction by setting the `PERL_DESTRUCT_LEVEL` environment variable, presuming `-DDEBUGGING` was enabled during perl build time. See [PERL_DESTRUCT_LEVEL](#) for more information.

A more complete garbage collection strategy will be implemented at a future date.

In the meantime, the best solution is to create a non-recursive container class that holds a pointer to the self-referential data structure. Define a `DESTROY` method for the containing object's class that manually breaks the circularities in the self-referential structure.

SEE ALSO

A kinder, gentler tutorial on object-oriented programming in Perl can be found in [perltoot](#), [perlboot](#) and [perltooc](#). You should also check out [perlbot](#) for other object tricks, traps, and tips, as well as [perlmodlib](#) for some style guides on constructing both modules and classes.

NAME

perlop – Perl operators and precedence

SYNOPSIS

Perl operators have the following associativity and precedence, listed from highest precedence to lowest. Operators borrowed from C keep the same precedence relationship with each other, even where C's precedence is slightly screwy. (This makes learning Perl easier for C folks.) With very few exceptions, these all operate on scalar values only, not array values.

```

left      terms and list operators (leftward)
left      ->
nonassoc  ++ --
right     **
right     ! ~ \ and unary + and -
left      =~ !~
left      * / % x
left      + - .
left      << >>
nonassoc  named unary operators
nonassoc  < > <= >= lt gt le ge
nonassoc  == != <=> eq ne cmp
left      &
left      | ^
left      &&
left      ||
nonassoc  .. ...
right     ?:
right     = += -= *= etc.
left      , =>
nonassoc  list operators (rightward)
right     not
left      and
left      or xor

```

In the following sections, these operators are covered in precedence order.

Many operators can be overloaded for objects. See [overload](#).

DESCRIPTION**Terms and List Operators (Leftward)**

A TERM has the highest precedence in Perl. They include variables, quote and quote-like operators, any expression in parentheses, and any function whose arguments are parenthesized. Actually, there aren't really functions in this sense, just list operators and unary operators behaving as functions because you put parentheses around the arguments. These are all documented in [perlfunc](#).

If any list operator (`print()`, etc.) or any unary operator (`chdir()`, etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call.

In the absence of parentheses, the precedence of list operators such as `print`, `sort`, or `chmod` is either very high or very low depending on whether you are looking at the left side or the right side of the operator. For example, in

```

@ary = (1, 3, sort 4, 2);
print @ary;           # prints 1324

```

the commas on the right of the `sort` are evaluated before the `sort`, but the commas on the left are evaluated

after. In other words, list operators tend to gobble up all arguments that follow, and then act like a simple TERM with regard to the preceding expression. Be careful with parentheses:

```
# These evaluate exit before doing the print:
print($foo, exit); # Obviously not what you want.
print $foo, exit;  # Nor is this.

# These do the print before evaluating exit:
(print $foo), exit; # This is what you want.
print($foo), exit; # Or this.
print ($foo), exit; # Or even this.
```

Also note that

```
print ($foo & 255) + 1, "\n";
```

probably doesn't do what you expect at first glance. See [Named Unary Operators](#) for more discussion of this.

Also parsed as terms are the `do { }` and `eval { }` constructs, as well as subroutine and method calls, and the anonymous constructors `[]` and `{ }`.

See also [Quote and Quote-like Operators](#) toward the end of this section, as well as [O Operators](#)".

The Arrow Operator

"< -" is an infix dereference operator, just as it is in C and C++. If the right side is either a `[...]`, `{...}`, or a `(...)` subscript, then the left side must be either a hard or symbolic reference to an array, a hash, or a subroutine respectively. (Or technically speaking, a location capable of holding a hard reference, if it's an array or hash reference being used for assignment.) See [perlreftut](#) and [perlref](#).

Otherwise, the right side is a method name or a simple scalar variable containing either the method name or a subroutine reference, and the left side must be either an object (a blessed reference) or a class name (that is, a package name). See [perlobj](#).

Auto-increment and Auto-decrement

"++" and "--" work as in C. That is, if placed before a variable, they increment or decrement the variable before returning the value, and if placed after, increment or decrement the variable after returning the value.

The auto-increment operator has a little extra builtin magic to it. If you increment a variable that is numeric, or that has ever been used in a numeric context, you get a normal increment. If, however, the variable has been used in only string contexts since it was set, and has a value that is not the empty string and matches the pattern `/^[a-zA-Z]*[0-9]*\z/`, the increment is done as a string, preserving each character within its range, with carry:

```
print ++($foo = '99');      # prints '100'
print ++($foo = 'a0');      # prints 'a1'
print ++($foo = 'Az');      # prints 'Ba'
print ++($foo = 'zz');      # prints 'aaa'
```

The auto-decrement operator is not magical.

Exponentiation

Binary `"**"` is the exponentiation operator. It binds even more tightly than unary minus, so `-2**4` is `-(2**4)`, not `(-2)**4`. (This is implemented using C's `pow(3)` function, which actually works on doubles internally.)

Symbolic Unary Operators

Unary `"!"` performs logical negation, i.e., "not". See also `not` for a lower precedence version of this.

Unary `"-"` performs arithmetic negation if the operand is numeric. If the operand is an identifier, a string consisting of a minus sign concatenated with the identifier is returned. Otherwise, if the string starts with a plus or minus, a string starting with the opposite sign is returned. One effect of these rules is that

`-bareword` is equivalent to `"-bareword"`.

Unary `~` performs bitwise negation, i.e., 1's complement. For example, `0666 & ~027` is `0640`. (See also *Integer Arithmetic* and *Bitwise String Operators*.) Note that the width of the result is platform-dependent: `~0` is 32 bits wide on a 32-bit platform, but 64 bits wide on a 64-bit platform, so if you are expecting a certain bit width, remember use the `&` operator to mask off the excess bits.

Unary `+` has no effect whatsoever, even on strings. It is useful syntactically for separating a function name from a parenthesized expression that would otherwise be interpreted as the complete list of function arguments. (See examples above under *Terms and List Operators (Leftward)*.)

Unary `\` creates a reference to whatever follows it. See *perlrefut* and *perlref*. Do not confuse this behavior with the behavior of backslash within a string, although both forms do convey the notion of protecting the next thing from interpolation.

Binding Operators

Binary `=~` binds a scalar expression to a pattern match. Certain operations search or modify the string `$_` by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or transliteration. The left argument is what is supposed to be searched, substituted, or transliterated instead of the default `$_`. When used in scalar context, the return value generally indicates the success of the operation. Behavior in list context depends on the particular operator. See for details.

If the right argument is an expression rather than a search pattern, substitution, or transliteration, it is interpreted as a search pattern at run time. This can be less efficient than an explicit search, because the pattern must be compiled every time the expression is evaluated.

Binary `!~` is just like `=~` except the return value is negated in the logical sense.

Multiplicative Operators

Binary `*` multiplies two numbers.

Binary `/` divides two numbers.

Binary `%` computes the modulus of two numbers. Given integer operands `$a` and `$b`: If `$b` is positive, then `$a % $b` is `$a` minus the largest multiple of `$b` that is not greater than `$a`. If `$b` is negative, then `$a % $b` is `$a` minus the smallest multiple of `$b` that is not less than `$a` (i.e. the result will be less than or equal to zero). Note that when `use integer` is in scope, `%` gives you direct access to the modulus operator as implemented by your C compiler. This operator is not as well defined for negative operands, but it will execute faster.

Binary `x` is the repetition operator. In scalar context or if the left operand is not enclosed in parentheses, it returns a string consisting of the left operand repeated the number of times specified by the right operand. In list context, if the left operand is enclosed in parentheses, it repeats the list.

```
print '-' x 80;           # print row of dashes
print "\t" x ($tab/8), ' ' x ($tab%8);    # tab over
@ones = (1) x 80;         # a list of 80 1's
@ones = (5) x @ones;      # set all elements to 5
```

Additive Operators

Binary `+` returns the sum of two numbers.

Binary `-` returns the difference of two numbers.

Binary `.` concatenates two strings.

Shift Operators

Binary `<<` returns the value of its left argument shifted left by the number of bits specified by the right argument. Arguments should be integers. (See also *Integer Arithmetic*.)

Binary `<<` returns the value of its left argument shifted right by the number of bits specified by the right argument. Arguments should be integers. (See also [Integer Arithmetic](#).)

Note that both `<<` and `>>` in Perl are implemented directly using `<<` and `>>` in C. If `use integer` (see [Integer Arithmetic](#)) is in force then signed C integers are used, else unsigned C integers are used. Either way, the implementation isn't going to generate results larger than the size of the integer type Perl was built with (32 bits or 64 bits).

The result of overflowing the range of the integers is undefined because it is undefined also in C. In other words, using 32-bit integers, `< 1 << 32` is undefined. Shifting by a negative number of bits is also undefined.

Named Unary Operators

The various named unary operators are treated as functions with one argument, with optional parentheses. These include the filetest operators, like `-f`, `-M`, etc. See [perlfunc](#).

If any list operator (`print()`, etc.) or any unary operator (`chdir()`, etc.) is followed by a left parenthesis as the next token, the operator and arguments within parentheses are taken to be of highest precedence, just like a normal function call. For example, because named unary operators are higher precedence than `||`:

```
chdir $foo      || die;      # (chdir $foo) || die
chdir($foo)    || die;      # (chdir $foo) || die
chdir ($foo)   || die;      # (chdir $foo) || die
chdir +($foo)  || die;      # (chdir $foo) || die
```

but, because `*` is higher precedence than named operators:

```
chdir $foo * 20;    # chdir ($foo * 20)
chdir($foo) * 20;   # (chdir $foo) * 20
chdir ($foo) * 20;  # (chdir $foo) * 20
chdir +($foo) * 20; # chdir ($foo * 20)

rand 10 * 20;       # rand (10 * 20)
rand(10) * 20;      # (rand 10) * 20
rand (10) * 20;     # (rand 10) * 20
rand +(10) * 20;    # rand (10 * 20)
```

See also ["Terms and List Operators \(Leftward\)"](#).

Relational Operators

Binary `<` returns true if the left argument is numerically less than the right argument.

Binary `>` returns true if the left argument is numerically greater than the right argument.

Binary `<=` returns true if the left argument is numerically less than or equal to the right argument.

Binary `=` returns true if the left argument is numerically greater than or equal to the right argument.

Binary `lt` returns true if the left argument is stringwise less than the right argument.

Binary `gt` returns true if the left argument is stringwise greater than the right argument.

Binary `le` returns true if the left argument is stringwise less than or equal to the right argument.

Binary `ge` returns true if the left argument is stringwise greater than or equal to the right argument.

Equality Operators

Binary `==` returns true if the left argument is numerically equal to the right argument.

Binary `!=` returns true if the left argument is numerically not equal to the right argument.

Binary `<=>` returns `-1`, `0`, or `1` depending on whether the left argument is numerically less than, equal to, or greater than the right argument. If your platform supports NaNs (not-a-numbers) as numeric values, using

them with "<=" returns undef. NaN is not "<", "==", "", "<=" or "=" anything (even NaN), so those 5 return false. NaN != NaN returns true, as does NaN != anything else. If your platform doesn't support NaNs then NaN is just a string with numeric value 0.

```
perl -le '$a = NaN; print "No NaN support here" if $a == $a'
perl -le '$a = NaN; print "NaN support here" if $a != $a'
```

Binary "eq" returns true if the left argument is stringwise equal to the right argument.

Binary "ne" returns true if the left argument is stringwise not equal to the right argument.

Binary "cmp" returns -1, 0, or 1 depending on whether the left argument is stringwise less than, equal to, or greater than the right argument.

"lt", "le", "ge", "gt" and "cmp" use the collation (sort) order specified by the current locale if use locale is in effect. See [perllocale](#).

Bitwise And

Binary "&" returns its operators ANDed together bit by bit. (See also [Integer Arithmetic](#) and [Bitwise String Operators](#).)

Bitwise Or and Exclusive Or

Binary "|" returns its operators ORed together bit by bit. (See also [Integer Arithmetic](#) and [Bitwise String Operators](#).)

Binary "^" returns its operators XORed together bit by bit. (See also [Integer Arithmetic](#) and [Bitwise String Operators](#).)

C-style Logical And

Binary "&&" performs a short-circuit logical AND operation. That is, if the left operand is false, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

C-style Logical Or

Binary "||" performs a short-circuit logical OR operation. That is, if the left operand is true, the right operand is not even evaluated. Scalar or list context propagates down to the right operand if it is evaluated.

The || and && operators differ from C's in that, rather than returning 0 or 1, they return the last value evaluated. Thus, a reasonably portable way to find out the home directory (assuming it's not "0") might be:

```
$home = $ENV{'HOME'} || $ENV{'LOGDIR'} ||
      (getpwuid($<))[7] || die "You're homeless!\n";
```

In particular, this means that you shouldn't use this for selecting between two aggregates for assignment:

```
@a = @b || @c;           # this is wrong
@a = scalar(@b) || @c;    # really meant this
@a = @b ? @b : @c;        # this works fine, though
```

As more readable alternatives to && and || when used for control flow, Perl provides and and or operators (see below). The short-circuit behavior is identical. The precedence of "and" and "or" is much lower, however, so that you can safely use them after a list operator without the need for parentheses:

```
unlink "alpha", "beta", "gamma"
    or gripe(), next LINE;
```

With the C-style operators that would have been written like this:

```
unlink("alpha", "beta", "gamma")
    || (gripe(), next LINE);
```

Using "or" for assignment is unlikely to do what you want; see below.

Range Operators

Binary `".."` is the range operator, which is really two different operators depending on the context. In list context, it returns an list of values counting (up by ones) from the left value to the right value. If the left value is greater than the right value then it returns the empty array. The range operator is useful for writing `foreach (1..10)` loops and for doing slice operations on arrays. In the current implementation, no temporary array is created when the range operator is used as the expression in `foreach` loops, but older versions of Perl might burn a lot of memory when you write something like this:

```
for (1 .. 1_000_000) {
    # code
}
```

The range operator also works on strings, using the magical auto-increment, see below.

In scalar context, `".."` returns a boolean value. The operator is bistable, like a flip-flop, and emulates the line-range (comma) operator of **sed**, **awk**, and various editors. Each `".."` operator maintains its own boolean state. It is false as long as its left operand is false. Once the left operand is true, the range operator stays true until the right operand is true, *AFTER* which the range operator becomes false again. It doesn't become false till the next time the range operator is evaluated. It can test the right operand and become false on the same evaluation it became true (as in **awk**), but it still returns true once. If you don't want it to test the right operand till the next evaluation, as in **sed**, just use three dots ("`...`") instead of two. In all other regards, `"..."` behaves just like `".."` does.

The right operand is not evaluated while the operator is in the "false" state, and the left operand is not evaluated while the operator is in the "true" state. The precedence is a little lower than `||` and `&&`. The value returned is either the empty string for false, or a sequence number (beginning with 1) for true. The sequence number is reset for each range encountered. The final sequence number in a range has the string "E0" appended to it, which doesn't affect its numeric value, but gives you something to search for if you want to exclude the endpoint. You can exclude the beginning point by waiting for the sequence number to be greater than 1. If either operand of scalar `".."` is a constant expression, that operand is implicitly compared to the `$_` variable, the current line number. Examples:

As a scalar operator:

```
if (101 .. 200) { print; } # print 2nd hundred lines
next line if (1 .. /^$/); # skip header lines
s/^/> / if (/^$/ .. eof()); # quote body

# parse mail messages
while (<>) {
    $in_header = 1 .. /^$/;
    $in_body   = /^$/ .. eof();
    # do something based on those
} continue {
    close ARGV if eof;          # reset $. each file
}
```

As a list operator:

```
for (101 .. 200) { print; } # print $_ 100 times
@foo = @foo[0 .. $#foo];    # an expensive no-op
@foo = @foo[$#foo-4 .. $#foo]; # slice last 5 items
```

The range operator (in list context) makes use of the magical auto-increment algorithm if the operands are strings. You can say

```
@alphabet = ('A' .. 'Z');
```

to get all normal letters of the English alphabet, or

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$num & 15];
```

to get a hexadecimal digit, or

```
@z2 = ('01' .. '31'); print $z2[$mday];
```

to get dates with leading zeros. If the final value specified is not in the sequence that the magical increment would produce, the sequence goes until the next value would be longer than the final value specified.

Conditional Operator

Ternary "?:" is the conditional operator, just as in C. It works much like an if-then-else. If the argument before the ? is true, the argument before the : is returned, otherwise the argument after the : is returned. For example:

```
printf "I have %d dog%s.\n", $n,
      ($n == 1) ? '' : "s";
```

Scalar or list context propagates downward into the 2nd or 3rd argument, whichever is selected.

```
$a = $ok ? $b : $c; # get a scalar
@a = $ok ? @b : @c; # get an array
$a = $ok ? @b : @c; # oops, that's just a count!
```

The operator may be assigned to if both the 2nd and 3rd arguments are legal lvalues (meaning that you can assign to them):

```
($a_or_b ? $a : $b) = $c;
```

Because this operator produces an assignable result, using assignments without parentheses will get you in trouble. For example, this:

```
$a % 2 ? $a += 10 : $a += 2
```

Really means this:

```
(( $a % 2 ) ? ( $a += 10 ) : $a) += 2
```

Rather than this:

```
($a % 2) ? ($a += 10) : ($a += 2)
```

That should probably be written more simply as:

```
$a += ($a % 2) ? 10 : 2;
```

Assignment Operators

"=" is the ordinary assignment operator.

Assignment operators work as in C. That is,

```
$a += 2;
```

is equivalent to

```
$a = $a + 2;
```

although without duplicating any side effects that dereferencing the lvalue might trigger, such as from tie(). Other assignment operators work similarly. The following are recognized:

**=	+=	*=	&=	<<=	&&=
	-=	/=	=	>>=	=
	.=	%=	^=		
		x=			

Although these are grouped by family, they all have the precedence of assignment.

Unlike in C, the scalar assignment operator produces a valid lvalue. Modifying an assignment is equivalent

to doing the assignment and then modifying the variable that was assigned to. This is useful for modifying a copy of something, like this:

```
($tmp = $global) =~ tr [A-Z] [a-z];
```

Likewise,

```
($a += 2) *= 3;
```

is equivalent to

```
$a += 2;
$a *= 3;
```

Similarly, a list assignment in list context produces the list of lvalues assigned to, and a list assignment in scalar context returns the number of elements produced by the expression on the right hand side of the assignment.

Comma Operator

Binary `","` is the comma operator. In scalar context it evaluates its left argument, throws that value away, then evaluates its right argument and returns that value. This is just like C's comma operator.

In list context, it's just the list argument separator, and inserts both its arguments into the list.

The `=` digraph is mostly just a synonym for the comma operator. It's useful for documenting arguments that come in pairs. As of release 5.001, it also forces any word to the left of it to be interpreted as a string.

List Operators (Rightward)

On the right side of a list operator, it has very low precedence, such that it controls all comma-separated expressions found there. The only operators with lower precedence are the logical operators `"and"`, `"or"`, and `"not"`, which may be used to evaluate calls to list operators without the need for extra parentheses:

```
open HANDLE, "filename"
    or die "Can't open: $!\n";
```

See also discussion of list operators in [Terms and List Operators \(Leftward\)](#).

Logical Not

Unary `"not"` returns the logical negation of the expression to its right. It's the equivalent of `"!"` except for the very low precedence.

Logical And

Binary `"and"` returns the logical conjunction of the two surrounding expressions. It's equivalent to `&&` except for the very low precedence. This means that it short-circuits: i.e., the right expression is evaluated only if the left expression is true.

Logical or and Exclusive Or

Binary `"or"` returns the logical disjunction of the two surrounding expressions. It's equivalent to `||` except for the very low precedence. This makes it useful for control flow

```
print FH $data                or die "Can't write to FH: $!";
```

This means that it short-circuits: i.e., the right expression is evaluated only if the left expression is false. Due to its precedence, you should probably avoid using this for assignment, only for control flow.

```
$a = $b or $c;                # bug: this is wrong
($a = $b) or $c;              # really means this
$a = $b || $c;                # better written this way
```

However, when it's a list-context assignment and you're trying to use `"||"` for control flow, you probably need `"or"` so that the assignment takes higher precedence.

```
@info = stat($file) || die;    # oops, scalar sense of stat!
@info = stat($file) or die;     # better, now @info gets its due
```

Then again, you could always use parentheses.

Binary "xor" returns the exclusive-OR of the two surrounding expressions. It cannot short circuit, of course.

C Operators Missing From Perl

Here is what C has that Perl doesn't:

unary & Address-of operator. (But see the "&" operator for taking a reference.)

unary * Dereference-address operator. (Perl's prefix dereferencing operators are typed: \$, @, %, and &.)

(TYPE) Type-casting operator.

Quote and Quote-like Operators

While we usually think of quotes as literal values, in Perl they function as operators, providing various kinds of interpolating and pattern matching capabilities. Perl provides customary quote characters for these behaviors, but also provides a way for you to choose your quote character for any of them. In the following table, a { } represents any pair of delimiters you choose.

Customary	Generic	Meaning	Interpolates
' '	q{ }	Literal	no
" "	qq{ }	Literal	yes
` `	qx{ }	Command	yes*
	qw{ }	Word list	no
//	m{ }	Pattern match	yes*
	qr{ }	Pattern	yes*
	s{ }{ }	Substitution	yes*
	tr{ }{ }	Transliteration	no (but see below)
<<EOF		here-doc	yes*

* unless the delimiter is ' '.

Non-bracketing delimiters use the same character fore and aft, but the four sorts of brackets (round, angle, square, curly) will all nest, which means that

```
q{foo{bar}baz}
```

is the same as

```
'foo{bar}baz'
```

Note, however, that this does not always work for quoting Perl code:

```
$s = q{ if($a eq "}") ... }; # WRONG
```

is a syntax error. The `Text::Balanced` module (from CPAN, and starting from Perl 5.8 part of the standard distribution) is able to do this properly.

There can be whitespace between the operator and the quoting characters, except when # is being used as the quoting character. `q#foo#` is parsed as the string `foo`, while `q #foo#` is the operator `q` followed by a comment. Its argument will be taken from the next line. This allows you to write:

```
s {foo} # Replace foo
   {bar} # with bar.
```

The following escape sequences are available in constructs that interpolate and in transliterations.

\t	tab	(HT, TAB)
\n	newline	(NL)
\r	return	(CR)
\f	form feed	(FF)
\b	backspace	(BS)
\a	alarm (bell)	(BEL)


```

\e          escape          (ESC)
\033octal char(ESC)
\x1bhex char(ESC)
\x{263a}    wide hex char   (SMILEY)
\c[         control char    (ESC)
\N{name}    named Unicode character

```

The following escape sequences are available in constructs that interpolate but not in transliterations.

```

\l          lowercase next char
\u          uppercase next char
\L          lowercase till \E
\U          uppercase till \E
\E          end case modification
\Q          quote non-word characters till \E

```

If use locale is in effect, the case map used by \l, \L, \u and \U is taken from the current locale. See [perllocale](#). If Unicode (for example, \N{ } or wide hex characters of 0x100 or beyond) is being used, the case map used by \l, \L, \u and \U is as defined by Unicode. For documentation of \N{name}, see [charnames](#).

All systems use the virtual "\n" to represent a line terminator, called a "newline". There is no such thing as an unvarying, physical newline character. It is only an illusion that the operating system, device drivers, C libraries, and Perl all conspire to preserve. Not all systems read "\r" as ASCII CR and "\n" as ASCII LF. For example, on a Mac, these are reversed, and on systems without line terminator, printing "\n" may emit no actual data. In general, use "\n" when you mean a "newline" for your system, but use the literal ASCII when you need an exact character. For example, most networking protocols expect and prefer a CR+LF ("\015\012" or "\cM\cJ") for line terminators, and although they often accept just "\012", they seldom tolerate just "\015". If you get in the habit of using "\n" for networking, you may be burned some day.

For constructs that do interpolate, variables beginning with "\$" or "@" are interpolated. Subscripted variables such as \$a[3] or \$href-{key}[0] are also interpolated, as are array and hash slices. But method calls such as \$obj-meth are not.

Interpolating an array or slice interpolates the elements in order, separated by the value of \$", so is equivalent to interpolating join \$", @array. "Punctuation" arrays such as @+ are only interpolated if the name is enclosed in braces @{+}.

You cannot include a literal \$ or @ within a \Q sequence. An unescaped \$ or @ interpolates the corresponding variable, while escaping will cause the literal string \\$ to be inserted. You'll need to write something like m/\Quser\E@\Qhost/.

Patterns are subject to an additional level of interpretation as a regular expression. This is done as a second pass, after variables are interpolated, so that regular expressions may be incorporated into the pattern from the variables. If this is not what you want, use \Q to interpolate a variable literally.

Apart from the behavior described above, Perl does not expand multiple levels of interpolation. In particular, contrary to the expectations of shell programmers, back-quotes do *NOT* interpolate within double quotes, nor do single quotes impede evaluation of variables when used within double quotes.

Regexp Quote-Like Operators

Here are the quote-like operators that apply to pattern matching and related activities.

?PATTERN?

This is just like the /pattern/ search, except that it matches only once between calls to the reset() operator. This is a useful optimization when you want to see only the first occurrence of something in each file of a set of files, for instance. Only ?? patterns local to the current package are reset.

```

while (<>) {
    if (?^$?) {
        # blank line between header and body
    }
    continue {
        reset if eof;      # clear ?? status for next file
    }
}

```

This usage is vaguely deprecated, which means it just might possibly be removed in some distant future version of Perl, perhaps somewhere around the year 2168.

`m/PATTERN/cgimosx`
`/PATTERN/cgimosx`

Searches a string for a pattern match, and in scalar context returns true if it succeeds, false if it fails. If no string is specified via the `=~` or `!~` operator, the `$_` string is searched. (The string specified with `=~` need not be an lvalue—it may be the result of an expression evaluation, but remember the `=~` binds rather tightly.) See also [perlre](#). See [perllocale](#) for discussion of additional considerations that apply when use `locale` is in effect.

Options are:

```

c    Do not reset search position on a failed match when /g is in effect.
g    Match globally, i.e., find all occurrences.
i    Do case-insensitive pattern matching.
m    Treat string as multiple lines.
o    Compile pattern only once.
s    Treat string as single line.
x    Use extended regular expressions.

```

If `/` is the delimiter then the initial `m` is optional. With the `m` you can use any pair of non-alphanumeric, non-whitespace characters as delimiters. This is particularly useful for matching path names that contain `/`, to avoid LTS (leaning toothpick syndrome). If `"` is the delimiter, then the match-only-once rule of `?PATTERN?` applies. If `'` is the delimiter, no interpolation is performed on the PATTERN.

PATTERN may contain variables, which will be interpolated (and the pattern recompiled) every time the pattern search is evaluated, except for when the delimiter is a single quote. (Note that `$ (`, `$)`, and `$ |` are not interpolated because they look like end-of-string tests.) If you want such a pattern to be compiled only once, add a `/o` after the trailing delimiter. This avoids expensive run-time recompilations, and is useful when the value you are interpolating won't change over the life of the script. However, mentioning `/o` constitutes a promise that you won't change the variables in the pattern. If you change them, Perl won't even notice. See also [imosx](#)".

If the PATTERN evaluates to the empty string, the last *successfully* matched regular expression is used instead. In this case, only the `g` and `c` flags on the empty pattern is honoured – the other flags are taken from the original pattern. If no match has previously succeeded, this will (silently) act instead as a genuine empty pattern (which will always match).

If the `/g` option is not used, `m//` in list context returns a list consisting of the subexpressions matched by the parentheses in the pattern, i.e., `($1, $2, $3 . . .)`. (Note that here `$1` etc. are also set, and that this differs from Perl 4's behavior.) When there are no parentheses in the pattern, the return value is the list `(1)` for success. With or without parentheses, an empty list is returned upon failure.

Examples:

```

open(TTY, '/dev/tty');
<TTY> =~ /^Y/i && foo();    # do foo if desired

```

```

if (/Version: *([0-9.]*)/) { $version = $1; }
next if m#^/usr/spool/uucp#;

# poor man's grep
$args = shift;
while (<>) {
    print if /$arg/o;      # compile only once
}

if (($F1, $F2, $Etc) = ($foo =~ /^(\S+)\s+(\S+)\s*(.*)/))

```

This last example splits `$foo` into the first two words and the remainder of the line, and assigns those three fields to `$F1`, `$F2`, and `$Etc`. The conditional is true if any variables were assigned, i.e., if the pattern matched.

The `/g` modifier specifies global pattern matching—that is, matching as many times as possible within the string. How it behaves depends on the context. In list context, it returns a list of the substrings matched by any capturing parentheses in the regular expression. If there are no parentheses, it returns a list of all the matched strings, as if there were parentheses around the whole pattern.

In scalar context, each execution of `m//g` finds the next match, returning true if it matches, and false if there is no further match. The position after the last match can be read or set using the `pos()` function; see [pos](#). A failed match normally resets the search position to the beginning of the string, but you can avoid that by adding the `/c` modifier (e.g. `m//gc`). Modifying the target string also resets the search position.

You can intermix `m//g` matches with `m/\G.../g`, where `\G` is a zero-width assertion that matches the exact position where the previous `m//g`, if any, left off. Without the `/g` modifier, the `\G` assertion still anchors at `pos()`, but the match is of course only attempted once. Using `\G` without `/g` on a target string that has not previously had a `/g` match applied to it is the same as using the `\A` assertion to match the beginning of the string. Note also that, currently, `\G` is only properly supported when anchored at the very beginning of the pattern.

Examples:

```

# list context
($one,$five,$fifteen) = ('uptime' =~ /(\d+\.\d+)/g);

# scalar context
$/ = "";
while (defined($paragraph = <>)) {
    while ($paragraph =~ /[a-z] ['"]* [.!?]+ ['"]*\s/g) {
        $sentences++;
    }
}
print "$sentences\n";

# using m//gc with \G
$_ = "ppooqppqq";
while ($i++ < 2) {
    print "1: '";
    print $1 while /(o)/gc; print "'", pos="," pos, "\n";
    print "2: '";
    print $1 if /\G(q)/gc; print "'", pos="," pos, "\n";
    print "3: '";
    print $1 while /(p)/gc; print "'", pos="," pos, "\n";
}
print "Final: '$1', pos=","pos,""\n" if /\G(.)/;

```

The last example should print:

```
1: 'oo', pos=4
2: 'q', pos=5
3: 'pp', pos=7
1: '', pos=7
2: 'q', pos=8
3: '', pos=8
Final: 'q', pos=8
```

Notice that the final match matched `q` instead of `p`, which a match without the `\G` anchor would have done. Also note that the final match did not update `pos` — `pos` is only updated on a `/g` match. If the final match did indeed match `p`, it's a good bet that you're running an older (pre-5.6.0) Perl.

A useful idiom for lex-like scanners is `/\G.../gc`. You can combine several regexps like this to process a string part-by-part, doing different actions depending on which regexp matched. Each regexp tries to match where the previous one leaves off.

```
$_ = <<'EOL';
    $url = new URI::URL "http://www/";    die if $url eq "xXx";
EOL
LOOP:
{
    print(" digits"),      redo LOOP if /\G\d+\b[,.;]?\s*/gc;
    print(" lowercase"),    redo LOOP if /\G[a-z]+\b[,.;]?\s*/gc;
    print(" UPPERCASE"),    redo LOOP if /\G[A-Z]+\b[,.;]?\s*/gc;
    print(" Capitalized"),  redo LOOP if /\G[A-Z][a-z]+\b[,.;]?\s*/gc;
    print(" MiXeD"),        redo LOOP if /\G[A-Za-z]+\b[,.;]?\s*/gc;
    print(" alphanumeric"), redo LOOP if /\G[A-Za-z0-9]+\b[,.;]?\s*/gc;
    print(" line-noise"),   redo LOOP if /\G[^A-Za-z0-9]+/gc;
    print ". That's all!\n";
}
```

Here is the output (split into several lines):

```
line-noise lowercase line-noise lowercase UPPERCASE line-noise
UPPERCASE line-noise lowercase line-noise lowercase line-noise
lowercase lowercase line-noise lowercase lowercase line-noise
MiXeD line-noise. That's all!
```

q/STRING/
'STRING'

A single-quoted, literal string. A backslash represents a backslash unless followed by the delimiter or another backslash, in which case the delimiter or backslash is interpolated.

```
$foo = q!I said, "You said, 'She said it.'!";
$bar = q('This is it.');
```

\$baz = '\n'; # a two-character string

qq/STRING/
"STRING"

A double-quoted, interpolated string.

```
$_ .= qq
    (** The previous line contains the naughty word "$1".\n)
    if /\b(tc|java|python)\b/i; # :-)
$baz = "\n"; # a one-character string
```

qr/STRING/imosx

This operator quotes (and possibly compiles) its *STRING* as a regular expression. *STRING* is interpolated the same way as *PATTERN* in `m/PATTERN/`. If `"` is used as the delimiter, no interpolation is done. Returns a Perl value which may be used instead of the corresponding `/STRING/imosx` expression.

For example,

```
$rex = qr/my.STRING/is;
s/$rex/foo/;
```

is equivalent to

```
s/my.STRING/foo/is;
```

The result may be used as a subpattern in a match:

```
$re = qr/$pattern/;
$string =~ /foo${re}bar/; # can be interpolated in other patterns
$string =~ $re;          # or used standalone
$string =~ /$re/;        # or this way
```

Since Perl may compile the pattern at the moment of execution of `qr()` operator, using `qr()` may have speed advantages in some situations, notably if the result of `qr()` is used standalone:

```
sub match {
    my $patterns = shift;
    my @compiled = map qr/$_/i, @$patterns;
    grep {
        my $success = 0;
        foreach my $pat (@compiled) {
            $success = 1, last if /$pat/;
        }
        $success;
    } @_;
}
```

Precompilation of the pattern into an internal representation at the moment of `qr()` avoids a need to recompile the pattern every time a match `/ $pat /` is attempted. (Perl has many other internal optimizations, but none would be triggered in the above example if we did not use `qr()` operator.)

Options are:

- `i` Do case-insensitive pattern matching.
- `m` Treat string as multiple lines.
- `o` Compile pattern only once.
- `s` Treat string as single line.
- `x` Use extended regular expressions.

See [perlre](#) for additional information on valid syntax for *STRING*, and for a detailed look at the semantics of regular expressions.

qx/STRING/

'STRING' A string which is (possibly) interpolated and then executed as a system command with `/bin/sh` or its equivalent. Shell wildcards, pipes, and redirections will be honored. The collected standard output of the command is returned; standard error is unaffected. In scalar context, it comes back as a single (potentially multi-line) string, or `undef` if the command failed. In list context, returns a list of lines (however you've defined lines with `$/` or `$INPUT_RECORD_SEPARATOR`), or an empty list if the command failed.

Because backticks do not affect standard error, use shell file descriptor syntax (assuming the shell supports this) if you care to address this. To capture a command's STDERR and STDOUT together:

```
$output = `cmd 2>&1`;
```

To capture a command's STDOUT but discard its STDERR:

```
$output = `cmd 2>/dev/null`;
```

To capture a command's STDERR but discard its STDOUT (ordering is important here):

```
$output = `cmd 2>&1 1>/dev/null`;
```

To exchange a command's STDOUT and STDERR in order to capture the STDERR but leave its STDOUT to come out the old STDERR:

```
$output = `cmd 3>&1 1>&2 2>&3 3>&-`;
```

To read both a command's STDOUT and its STDERR separately, it's easiest and safest to redirect them separately to files, and then read from those files when the program is done:

```
system("program args 1>/tmp/program.stdout 2>/tmp/program.stderr");
```

Using single-quote as a delimiter protects the command from Perl's double-quote interpolation, passing it on to the shell instead:

```
$perl_info = qx(ps $$);           # that's Perl's $$
$shell_info = qx'ps $$';          # that's the new shell's $$
```

How that string gets evaluated is entirely subject to the command interpreter on your system. On most platforms, you will have to protect shell metacharacters if you want them treated literally. This is in practice difficult to do, as it's unclear how to escape which characters. See [perlsec](#) for a clean and safe example of a manual `fork()` and `exec()` to emulate backticks safely.

On some platforms (notably DOS-like ones), the shell may not be capable of dealing with multiline commands, so putting newlines in the string may not get you what you want. You may be able to evaluate multiple commands in a single line by separating them with the command separator character, if your shell supports that (e.g. `;` on many Unix shells; `&` on the Windows NT cmd shell).

Beginning with v5.6.0, Perl will attempt to flush all files opened for output before starting the child process, but this may not be supported on some platforms (see [perlport](#)). To be safe, you may need to set `$|` (`$AUTOFLUSH` in English) or call the `autoflush()` method of `IO::Handle` on any open handles.

Beware that some command shells may place restrictions on the length of the command line. You must ensure your strings don't exceed this limit after any necessary interpolations. See the platform-specific release notes for more details about your particular environment.

Using this operator can lead to programs that are difficult to port, because the shell commands called vary between systems, and may in fact not be present at all. As one example, the `type` command under the POSIX shell is very different from the `type` command under DOS. That doesn't mean you should go out of your way to avoid backticks when they're the right way to get something done. Perl was made to be a glue language, and one of the things it glues together is commands. Just understand what you're getting yourself into.

See [O Operators](#) for more discussion.

qw/STRING/

Evaluates to a list of the words extracted out of `STRING`, using embedded whitespace as the word delimiters. It can be understood as being roughly equivalent to:

```
split(' ', q/STRING/);
```

the difference being that it generates a real list at compile time. So this expression:

```
qw(foo bar baz)
```

is semantically equivalent to the list:

```
'foo', 'bar', 'baz'
```

Some frequently seen examples:

```
use POSIX qw( setlocale localeconv )
@EXPORT = qw( foo bar baz );
```

A common mistake is to try to separate the words with comma or to put comments into a multi-line `qw`-string. For this reason, the `use warnings` pragma and the `-w` switch (that is, the `$^W` variable) produces warnings if the `STRING` contains the `,` or the `#` character.

s/PATTERN/REPLACEMENT/egimosx

Searches a string for a pattern, and if found, replaces that pattern with the replacement text and returns the number of substitutions made. Otherwise it returns false (specifically, the empty string).

If no string is specified via the `=~` or `!~` operator, the `$_` variable is searched and modified. (The string specified with `=~` must be scalar variable, an array element, a hash element, or an assignment to one of those, i.e., an lvalue.)

If the delimiter chosen is a single quote, no interpolation is done on either the `PATTERN` or the `REPLACEMENT`. Otherwise, if the `PATTERN` contains a `$` that looks like a variable rather than an end-of-string test, the variable will be interpolated into the pattern at run-time. If you want the pattern compiled only once the first time the variable is interpolated, use the `/o` option. If the pattern evaluates to the empty string, the last successfully executed regular expression is used instead. See [perlre](#) for further explanation on these. See [perllocale](#) for discussion of additional considerations that apply when `use locale` is in effect.

Options are:

```
e  Evaluate the right side as an expression.
g  Replace globally, i.e., all occurrences.
i  Do case-insensitive pattern matching.
m  Treat string as multiple lines.
o  Compile pattern only once.
s  Treat string as single line.
x  Use extended regular expressions.
```

Any non-alphanumeric, non-whitespace delimiter may replace the slashes. If single quotes are used, no interpretation is done on the replacement string (the `/e` modifier overrides this, however). Unlike Perl 4, Perl 5 treats backticks as normal delimiters; the replacement text is not evaluated as a command. If the `PATTERN` is delimited by bracketing quotes, the `REPLACEMENT` has its own pair of quotes, which may or may not be bracketing quotes, e.g., `s(foo)(bar)` or `< s<foo/bar/`. A `/e` will cause the replacement portion to be treated as a full-fledged Perl expression and evaluated right then and there. It is, however, syntax checked at compile-time. A second `e` modifier will cause the replacement portion to be eval'd before being run as a Perl expression.

Examples:

```
s/\bgreen\b/mauve/g;           # don't change wintergreen
$path =~ s|/usr/bin|/usr/local/bin|;
```

```

s/Login: $foo/Login: $bar/; # run-time pattern

($foo = $bar) =~ s/this/that/; # copy first, then change

$count = ($paragraph =~ s/Mister\b/Mr./g); # get change-count

$_ = 'abc123xyz';
s/\d+/$&*2/e; # yields 'abc246xyz'
s/\d+/sprintf("%5d",$&)/e; # yields 'abc 246xyz'
s/\w/$& x 2/eg; # yields 'aabbcc 224466xxxyzz'

s/%(.)/$percent{$1}/g; # change percent escapes; no /e
s/%(.)/$percent{$1} || $&/ge; # expr now, so /e
s/^(\\w+)/&pod($1)/ge; # use function call

# expand variables in $_, but dynamics only, using
# symbolic dereferencing
s/\\$(\\w+)/${$1}/g;

# Add one to the value of any numbers in the string
s/(\\d+)/1 + $1/eg;

# This will expand any embedded scalar variable
# (including lexicals) in $_ : First $1 is interpolated
# to the variable name, and then evaluated
s/(\\$\\w+)/$1/eeg;

# Delete (most) C comments.
$program =~ s {
    /\*      # Match the opening delimiter.
    .*?     # Match a minimal number of characters.
    \*/      # Match the closing delimiter.
} []gsx;

s/^\s*(.*?)\s*$/$1/; # trim white space in $_, expensively

for ($variable) { # trim white space in $variable, cheap
    s/^\s+//;
    s/\s+$//;
}

s/([^\s]*) *([^\s]*)/$2 $1/; # reverse 1st two fields

```

Note the use of `$` instead of `\` in the last example. Unlike **sed**, we use the `<digit` form in only the left hand side. Anywhere else it's `$<digit`.

Occasionally, you can't use just a `/g` to get all the changes to occur that you might want. Here are two common cases:

```

# put commas in the right places in an integer
1 while s/(\\d)(\\d\\d\\d)(?!\\d)/$1,$2/g;

# expand tabs to 8-column spacing
1 while s/\\t+/' ' x (length($&)*8 - length($`)%8)/e;

```

tr/SEARCHLIST/REPLACEMENTLIST/cds

y/SEARCHLIST/REPLACEMENTLIST/cds

Transliterates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the `=~` or `!~` operator, the `$_` string is transliterated. (The string specified with `=~` must be a scalar variable, an array element, a hash element, or an assignment to one of those, i.e., an lvalue.)

A character range may be specified with a hyphen, so `tr/A-J/0-9/` does the same replacement as `tr/ACEGIBDFHJ/0246813579/`. For **sed** devotees, `y` is provided as a synonym for `tr`. If the SEARCHLIST is delimited by bracketing quotes, the REPLACEMENTLIST has its own pair of quotes, which may or may not be bracketing quotes, e.g., `tr [A-Z] [a-z]` or `tr (+\-* /) /ABCD/`.

Note that `tr` does **not** do regular expression character classes such as `\d` or `[:lower:]`. The `<tr` operator is not equivalent to the `tr(1)` utility. If you want to map strings between lower/upper cases, see [lc](#) and [uc](#), and in general consider using the `s` operator if you need regular expressions.

Note also that the whole range idea is rather unportable between character sets—and even within character sets they may cause results you probably didn’t expect. A sound principle is to use only ranges that begin from and end at either alphabets of equal case (a–e, A–E), or digits (0–4). Anything else is unsafe. If in doubt, spell out the character sets in full.

Options:

- `c` Complement the SEARCHLIST.
- `d` Delete found but unreplaced characters.
- `s` Squash duplicate replaced characters.

If the `/c` modifier is specified, the SEARCHLIST character set is complemented. If the `/d` modifier is specified, any characters specified by SEARCHLIST not found in REPLACEMENTLIST are deleted. (Note that this is slightly more flexible than the behavior of some **tr** programs, which delete anything they find in the SEARCHLIST, period.) If the `/s` modifier is specified, sequences of characters that were transliterated to the same character are squashed down to a single instance of the character.

If the `/d` modifier is used, the REPLACEMENTLIST is always interpreted exactly as specified. Otherwise, if the REPLACEMENTLIST is shorter than the SEARCHLIST, the final character is replicated till it is long enough. If the REPLACEMENTLIST is empty, the SEARCHLIST is replicated. This latter is useful for counting characters in a class or for squashing character sequences in a class.

Examples:

```
$ARGV[1] =~ tr/A-Z/a-z/;      # canonicalize to lower case
$cnt = tr/*/*/;              # count the stars in $_
$cnt = $sky =~ tr/*/*/;      # count the stars in $sky
$cnt = tr/0-9//;             # count the digits in $_
tr/a-zA-Z//s;                # bookkeeper -> bokeper
($HOST = $host) =~ tr/a-z/A-Z/;
tr/a-zA-Z/ /cs;              # change non-alphas to single space
tr [\200-\377]
  [\000-\177];               # delete 8th bit
```

If multiple transliterations are given for a character, only the first one is used:

```
tr/AAA/XYZ/
```

will transliterate any A to X.

Because the transliteration table is built at compile time, neither the SEARCHLIST nor the REPLACEMENTLIST are subjected to double quote interpolation. That means that if you want to use variables, you must use an `eval()`:

```
eval "tr/$oldlist/$newlist/";
die $@ if $@;

eval "tr/$oldlist/$newlist/, 1" or die $@;
```

<<EOF A line-oriented form of quoting is based on the shell "here-document" syntax. Following a `<<` you specify a string to terminate the quoted material, and all lines following the current line down to the terminating string are the value of the item. The terminating string may be either an identifier (a word), or some quoted text. If quoted, the type of quotes you use determines the treatment of the text, just as in regular quoting. An unquoted identifier works like double quotes. There must be no space between the `<<` and the identifier, unless the identifier is quoted. (If you put a space it will be treated as a null identifier, which is valid, and matches the first empty line.) The terminating string must appear by itself (unquoted and with no surrounding whitespace) on the terminating line.

```
print <<EOF;
The price is $Price.
EOF

print << "EOF"; # same as above
The price is $Price.
EOF

print << `EOC`; # execute commands
echo hi there
echo lo there
EOC

print <<"foo", <<"bar"; # you can stack them
I said foo.
foo
I said bar.
bar

myfunc(<< "THIS", 23, <<'THAT');
Here's a line
or two.
THIS
and here's another.
THAT
```

Just don't forget that you have to put a semicolon on the end to finish the statement, as Perl doesn't know you're not going to try to do this:

```
print <<ABC
179231
ABC
+ 20;
```

If you want your here-docs to be indented with the rest of the code, you'll need to remove leading whitespace from each line manually:

```
($quote = <<'FINIS') =~ s/^\s+//gm;
    The Road goes ever on and on,
    down from the door where it began.
FINIS
```

If you use a here-doc within a delimited construct, such as in `s//eg`, the quoted material must come on the lines following the final delimiter. So instead of

```
s/this/<<E . 'that'
the other
E
. 'more '/eg;
```

you have to write

```
s/this/<<E . 'that'
. 'more '/eg;
the other
E
```

If the terminating identifier is on the last line of the program, you must be sure there is a newline after it; otherwise, Perl will give the warning **Can't find string terminator "END" anywhere before EOF...**

Additionally, the quoting rules for the identifier are not related to Perl's quoting rules — `q()`, `qq()`, and the like are not supported in place of `'` and `"`, and the only interpolation is for backslashing the quoting character:

```
print << "abc\"def";
testing...
abc"def
```

Finally, quoted strings cannot span multiple lines. The general rule is that the identifier must be a string literal. Stick with that, and you should be safe.

Gory details of parsing quoted constructs

When presented with something that might have several different interpretations, Perl uses the **DWIM** (that's "Do What I Mean") principle to pick the most probable interpretation. This strategy is so successful that Perl programmers often do not suspect the ambivalence of what they write. But from time to time, Perl's notions differ substantially from what the author honestly meant.

This section hopes to clarify how Perl handles quoted constructs. Although the most common reason to learn this is to unravel labyrinthine regular expressions, because the initial steps of parsing are the same for all quoting operators, they are all discussed together.

The most important Perl parsing rule is the first one discussed below: when processing a quoted construct, Perl first finds the end of that construct, then interprets its contents. If you understand this rule, you may skip the rest of this section on the first reading. The other rules are likely to contradict the user's expectations much less frequently than this first one.

Some passes discussed below are performed concurrently, but because their results are the same, we consider them individually. For different quoting constructs, Perl performs different numbers of passes, from one to five, but these passes are always performed in the same order.

Finding the end

The first pass is finding the end of the quoted construct, whether it be a multicharacter delimiter `"\nEOF\n"` in the `<<EOF` construct, a `/` that terminates a `qq//` construct, a `]` which terminates a `qq[]` construct, or a `<` which terminates a fileglob started with `< <`.

When searching for single-character non-pairing delimiters, such as `/`, combinations of `\\` and `\/` are skipped. However, when searching for single-character pairing delimiter like `[`, combinations of `\\`, `\]`, and `\[` are all skipped, and nested `[`, `]` are skipped as well. When searching for multicharacter delimiters, nothing is skipped.

For constructs with three-part delimiters (`s///`, `y///`, and `tr///`), the search is repeated once more.

During this search no attention is paid to the semantics of the construct. Thus:

```
"$hash{"$foo/$bar"}"
```

or:

```
m/
    bar          # NOT a comment, this slash / terminated m//!
/x
```

do not form legal quoted expressions. The quoted part ends on the first " and /, and the rest happens to be a syntax error. Because the slash that terminated m// was followed by a SPACE, the example above is not m//x, but rather m// with no /x modifier. So the embedded # is interpreted as a literal #.

Removal of backslashes before delimiters

During the second pass, text between the starting and ending delimiters is copied to a safe location, and the \ is removed from combinations consisting of \ and delimiter—or delimiters, meaning both starting and ending delimiters will should these differ. This removal does not happen for multi-character delimiters. Note that the combination \\ is left intact, just as it was.

Starting from this step no information about the delimiters is used in parsing.

Interpolation

The next step is interpolation in the text obtained, which is now delimiter-independent. There are four different cases.

```
<<'EOF', m'', s'', tr///, y///
```

No interpolation is performed.

```
'', q//
```

The only interpolation is removal of \ from pairs \\.

```
"", '', qq//, qx//, < <file*glob
```

\Q, \U, \u, \L, \l (possibly paired with \E) are converted to corresponding Perl constructs. Thus, "\$foo\Qbaz\$bar" is converted to \$foo . (quotemeta("baz" . \$bar)) internally. The other combinations are replaced with appropriate expansions.

Let it be stressed that *whatever falls between \Q and \E* is interpolated in the usual way. Something like "\Q\E" has no \E inside. instead, it has \Q, \\, and E, so the result is the same as for "\\\E". As a general rule, backslashes between \Q and \E may lead to counterintuitive results. So, "\Q\t\E" is converted to quotemeta("\t"), which is the same as "\\\t" (since TAB is not alphanumeric). Note also that:

```
$str = '\t';
return "\Q$str";
```

may be closer to the conjectural *intention* of the writer of "\Q\t\E".

Interpolated scalars and arrays are converted internally to the join and . catenation operations. Thus, "\$foo XXX \@arr" becomes:

```
$foo . " XXX '" . (join $", @arr) . "'";
```

All operations above are performed simultaneously, left to right.

Because the result of "\Q STRING \E" has all metacharacters quoted, there is no way to insert a literal \$ or @ inside a \Q\E pair. If protected by \, \$ will be quoted to become "\\\$"; if not, it is interpreted as the start of an interpolated scalar.

Note also that the interpolation code needs to make a decision on where the interpolated scalar ends. For instance, whether < "a \$b - {c}" really means:

```
"a " . $b . " -> {c}";
```

or:

```
"a " . $b -> {c};
```

Most of the time, the longest possible text that does not include spaces between components and which contains matching braces or brackets. because the outcome may be determined by voting based on heuristic estimators, the result is not strictly predictable. Fortunately, it's usually correct for ambiguous cases.

```
?RE?, /RE/, m/RE/, s/RE/fooo/,
```

Processing of `\Q`, `\U`, `\u`, `\L`, `\l`, and interpolation happens (almost) as with `qq//` constructs, but the substitution of `\` followed by RE-special chars (including `\`) is not performed. Moreover, inside `(?{BLOCK})`, `(?# comment)`, and a `#`-comment in a `//x`-regular expression, no processing is performed whatsoever. This is the first step at which the presence of the `//x` modifier is relevant.

Interpolation has several quirks: `$|`, `$(`, and `$)` are not interpolated, and constructs `$var[SOMETHING]` are voted (by several different estimators) to be either an array element or `$var` followed by an RE alternative. This is where the notation `${arr[$bar]}` comes handy: `/${arr[0-9]}/` is interpreted as array element `-9`, not as a regular expression from the variable `$arr` followed by a digit, which would be the interpretation of `/${arr[0-9]}/`. Since voting among different estimators may occur, the result is not predictable.

It is at this step that `\1` is begrudgingly converted to `$1` in the replacement text of `s///` to correct the incorrigible *sed* hackers who haven't picked up the saner idiom yet. A warning is emitted if the `use warnings` pragma or the `-w` command-line flag (that is, the `$^W` variable) was set.

The lack of processing of `\\` creates specific restrictions on the post-processed text. If the delimiter is `/`, one cannot get the combination `\/` into the result of this step. `/` will finish the regular expression, `\/` will be stripped to `/` on the previous step, and `\\/` will be left as is. Because `/` is equivalent to `\/` inside a regular expression, this does not matter unless the delimiter happens to be character special to the RE engine, such as in `s*foo*bar*`, `m[foo]`, or `?foo?`; or an alphanumeric char, as in:

```
m m ^ a \s* b mmx;
```

In the RE above, which is intentionally obfuscated for illustration, the delimiter is `m`, the modifier is `mx`, and after backslash-removal the RE is the same as for `m/ ^ a s* b /mx`. There's more than one reason you're encouraged to restrict your delimiters to non-alphanumeric, non-whitespace choices.

This step is the last one for all constructs except regular expressions, which are processed further.

Interpolation of regular expressions

Previous steps were performed during the compilation of Perl code, but this one happens at run time—although it may be optimized to be calculated at compile time if appropriate. After preprocessing described above, and possibly after evaluation if catenation, joining, casing translation, or metaquoting are involved, the resulting *string* is passed to the RE engine for compilation.

Whatever happens in the RE engine might be better discussed in [perlre](#), but for the sake of continuity, we shall do so here.

This is another step where the presence of the `//x` modifier is relevant. The RE engine scans the string from left to right and converts it to a finite automaton.

Backslashed characters are either replaced with corresponding literal strings (as with `\{`), or else they generate special nodes in the finite automaton (as with `\b`). Characters special to the RE engine (such as `|`) generate corresponding nodes or groups of nodes. `(?# . . .)` comments are ignored. All the rest is either converted to literal strings to match, or else is ignored (as is whitespace and `#`-style comments if `//x` is present).

Parsing of the bracketed character class construct, `[...]`, is rather different than the rule used for the rest of the pattern. The terminator of this construct is found using the same rules as for finding the terminator of a `{}`-delimited construct, the only exception being that `]` immediately following `[` is treated as though preceded by a backslash. Similarly, the terminator of `(?{...})` is found using the same rules as for finding the terminator of a `{}`-delimited construct.

It is possible to inspect both the string given to RE engine and the resulting finite automaton. See the arguments `debug/debugcolor` in the use [re](#) pragma, as well as Perl's **-Dr** command-line switch documented in [Command Switches in perlrun](#).

Optimization of regular expressions

This step is listed for completeness only. Since it does not change semantics, details of this step are not documented and are subject to change without notice. This step is performed over the finite automaton that was generated during the previous pass.

It is at this stage that `split()` silently optimizes `/^/` to mean `/^/m`.

I/O Operators

There are several I/O operators you should know about.

A string enclosed by backticks (grave accents) first undergoes double-quote interpolation. It is then interpreted as an external command, and the output of that command is the value of the backtick string, like in a shell. In scalar context, a single string consisting of all output is returned. In list context, a list of values is returned, one per line of output. (You can set `$/` to use a different line terminator.) The command is executed each time the pseudo-literal is evaluated. The status value of the command is returned in `$?` (see [perlvar](#) for the interpretation of `$?`). Unlike in **csh**, no translation is done on the return data—newlines remain newlines. Unlike in any of the shells, single quotes do not hide variable names in the command from interpretation. To pass a literal dollar-sign through to the shell you need to hide it with a backslash. The generalized form of backticks is `qx//`. (Because backticks always undergo shell expansion as well, see [perlsec](#) for security concerns.)

In scalar context, evaluating a filehandle in angle brackets yields the next line from that file (the newline, if any, included), or `undef` at end-of-file or on error. When `$/` is set to `undef` (sometimes known as file-slurp mode) and the file is empty, it returns `''` the first time, followed by `undef` subsequently.

Ordinarily you must assign the returned value to a variable, but there is one situation where an automatic assignment happens. If and only if the input symbol is the only thing inside the conditional of a `while` statement (even if disguised as a `for(;;)` loop), the value is automatically assigned to the global variable `$_`, destroying whatever was there previously. (This may seem like an odd thing to you, but you'll use the construct in almost every Perl script you write.) The `$_` variable is not implicitly localized. You'll have to put a `local $_;` before the loop if you want that to happen.

The following lines are equivalent:

```
while (defined($_ = <STDIN>)) { print; }
while ($_ = <STDIN>) { print; }
while (<STDIN>) { print; }
for (;<STDIN>;) { print; }
print while defined($_ = <STDIN>);
print while ($_ = <STDIN>);
print while <STDIN>;
```

This also behaves similarly, but avoids `$_`:

```
while (my $line = <STDIN>) { print $line }
```

In these loop constructs, the assigned value (whether assignment is automatic or explicit) is then tested to see whether it is defined. The defined test avoids problems where line has a string value that would be treated as false by Perl, for example a `""` or a `"0"` with no trailing newline. If you really mean for such values to terminate the loop, they should be tested for explicitly:

```
while (($_ = <STDIN>) ne '0') { ... }
while (<STDIN>) { last unless $_; ... }
```

In other boolean contexts, `< <filehandle` without an explicit defined test or comparison elicit a warning if the `use warnings` pragma or the `-w` command-line switch (the `$^W` variable) is in effect.

The filehandles `STDIN`, `STDOUT`, and `STDERR` are predefined. (The filehandles `stdin`, `stdout`, and `stderr` will also work except in packages, where they would be interpreted as local identifiers rather than global.) Additional filehandles may be created with the `open()` function, amongst others. See [perlopentut](#) and [open](#) for details on this.

If a `<FILEHANDLE` is used in a context that is looking for a list, a list comprising all input lines is returned, one line per list element. It's easy to grow to a rather large data space this way, so use with care.

`<FILEHANDLE` may also be spelled `readline(*FILEHANDLE)`. See [readline](#).

The null filehandle `<` is special: it can be used to emulate the behavior of `sed` and `awk`. Input from `<` comes either from standard input, or from each file listed on the command line. Here's how it works: the first time `<` is evaluated, the `@ARGV` array is checked, and if it is empty, `$ARGV[0]` is set to `"-"`, which when opened gives you standard input. The `@ARGV` array is then processed as a list of filenames. The loop

```
while (< >) {
    ...
    # code for each line
}
```

is equivalent to the following Perl-like pseudo code:

```
unshift(@ARGV, '-') unless @ARGV;
while ($ARGV = shift) {
    open(ARGV, $ARGV);
    while (<ARGV>) {
        ...
        # code for each line
    }
}
```

except that it isn't so cumbersome to say, and will actually work. It really does shift the `@ARGV` array and put the current filename into the `$ARGV` variable. It also uses filehandle `ARGV` internally—`<` is just a synonym for `<ARGV`, which is magical. (The pseudo code above doesn't work because it treats `<ARGV` as non-magical.)

You can modify `@ARGV` before the first `<` as long as the array ends up containing the list of filenames you really want. Line numbers (`$.`) continue as though the input were one big happy file. See the example in [eof](#) for how to reset line numbers on each file.

If you want to set `@ARGV` to your own list of files, go right ahead. This sets `@ARGV` to all plain text files if no `@ARGV` was given:

```
@ARGV = grep { -f && -T } glob('*') unless @ARGV;
```

You can even set them to pipe commands. For example, this automatically filters compressed arguments through **gzip**:

```
@ARGV = map { /\.(gz|Z)$/ ? "gzip -dc < $_ |" : $_ } @ARGV;
```

If you want to pass switches into your script, you can use one of the `Getopts` modules or put a loop on the front like this:

```
while ($_ = $ARGV[0], /^-/) {
    shift;
    last if /^--$/;
    if (/^-D(.*)/) { $debug = $1 }
    if (/^-v/)      { $verbose++ }
}
```

```

        # ...          # other switches
    }
    while (<>) {
        # ...          # code for each line
    }

```

The `<` symbol will return `undef` for end-of-file only once. If you call it again after this, it will assume you are processing another `@ARGV` list, and if you haven't set `@ARGV`, will read input from `STDIN`.

If what the angle brackets contain is a simple scalar variable (e.g., `<$foo>`), then that variable contains the name of the filehandle to input from, or its `typeglob`, or a reference to the same. For example:

```

$fh = \*STDIN;
$line = <$fh>;

```

If what's within the angle brackets is neither a filehandle nor a simple scalar variable containing a filehandle name, `typeglob`, or `typeglob` reference, it is interpreted as a filename pattern to be globbed, and either a list of filenames or the next filename in the list is returned, depending on context. This distinction is determined on syntactic grounds alone. That means `<$x` is always a `readline()` from an indirect handle, but `<$hash{key}` is always a `glob()`. That's because `$x` is a simple scalar variable, but `$hash{key}` is not—it's a hash element.

One level of double-quote interpretation is done first, but you can't say `<$foo>` because that's an indirect filehandle as explained in the previous paragraph. (In older versions of Perl, programmers would insert curly brackets to force interpretation as a filename glob: `<${foo}>`. These days, it's considered cleaner to call the internal function directly as `glob($foo)`, which is probably the right way to have done it in the first place.) For example:

```

while (<*.c>) {
    chmod 0644, $_;
}

```

is roughly equivalent to:

```

open(FOO, "echo *.c | tr -s ' \t\r\f' '\\012\\012\\012\\012' |");
while (<FOO>) {
    chomp;
    chmod 0644, $_;
}

```

except that the globbing is actually done internally using the standard `File::Glob` extension. Of course, the shortest way to do the above is:

```

chmod 0644, <*.c>;

```

A (file)glob evaluates its (embedded) argument only when it is starting a new list. All values must be read before it will start over. In list context, this isn't important because you automatically get them all anyway. However, in scalar context the operator returns the next value each time it's called, or `undef` when the list has run out. As with filehandle reads, an automatic `defined` is generated when the glob occurs in the test part of a `while`, because legal glob returns (e.g. a file called `)` would otherwise terminate the loop. Again, `undef` is returned only once. So if you're expecting a single value from a glob, it is much better to say

```

($file) = <blurch*>;

```

than

```

$file = <blurch*>;

```

because the latter will alternate between returning a filename and returning false.

If you're trying to do variable interpolation, it's definitely better to use the `glob()` function, because the older notation can cause people to become confused with the indirect filehandle notation.


```
@files = glob("$dir/*.ch");
@files = glob($files[$i]);
```

Constant Folding

Like C, Perl does a certain amount of expression evaluation at compile time whenever it determines that all arguments to an operator are static and have no side effects. In particular, string concatenation happens at compile time between literals that don't do variable substitution. Backslash interpolation also happens at compile time. You can say

```
'Now is the time for all' . "\n" .
'good men to come to.'
```

and this all reduces to one string internally. Likewise, if you say

```
foreach $file (@filenames) {
    if (-s $file > 5 + 100 * 2**16) { }
```

the compiler will precompute the number which that expression represents so that the interpreter won't have to.

Bitwise String Operators

Bitstrings of any size may be manipulated by the bitwise operators (`~` | `&` `^`).

If the operands to a binary bitwise op are strings of different sizes, `|` and `^` ops act as though the shorter operand had additional zero bits on the right, while the `&` op acts as though the longer operand were truncated to the length of the shorter. The granularity for such extension or truncation is one or more bytes.

```
# ASCII-based examples
print "j p \n" ^ " a h";           # prints "JAPH\n"
print "JA" | " ph\n";               # prints "japh\n"
print "japh\nJunk" & '_____';    # prints "JAPH\n"
print 'p N$' ^ " E<H\n";           # prints "Perl\n"
```

If you are intending to manipulate bitstrings, be certain that you're supplying bitstrings: If an operand is a number, that will imply a **numeric** bitwise operation. You may explicitly show which type of operation you intend by using `" "` or `0+`, as in the examples below.

```
$foo = 150 | 105 ;           # yields 255 (0x96 | 0x69 is 0xFF)
$foo = '150' | 105 ;         # yields 255
$foo = 150 | '105';          # yields 255
$foo = '150' | '105';        # yields string '155' (under ASCII)

$baz = 0+$foo & 0+$bar;      # both ops explicitly numeric
$biz = "$foo" ^ "$bar";      # both ops explicitly stringy
```

See [vec](#) for information on how to manipulate individual bits in a bit vector.

Integer Arithmetic

By default, Perl assumes that it must do most of its arithmetic in floating point. But by saying

```
use integer;
```

you may tell the compiler that it's okay to use integer operations (if it feels like it) from here to the end of the enclosing BLOCK. An inner BLOCK may countermand this by saying

```
no integer;
```

which lasts until the end of that BLOCK. Note that this doesn't mean everything is only an integer, merely that Perl may use integer operations if it is so inclined. For example, even under `use integer`, if you take the `sqrt(2)`, you'll still get 1.4142135623731 or so.

Used on numbers, the bitwise operators ("&", "|", "^", "~", "<<", and ">>") always produce integral results. (But see also [Bitwise String Operators](#).) However, `use integer` still has meaning for them. By default, their results are interpreted as unsigned integers, but if `use integer` is in effect, their results are interpreted as signed integers. For example, `~0` usually evaluates to a large integral value. However, `use integer; ~0` is `-1` on twos-complement machines.

Floating-point Arithmetic

While `use integer` provides integer-only arithmetic, there is no analogous mechanism to provide automatic rounding or truncation to a certain number of decimal places. For rounding to a certain number of digits, `sprintf()` or `printf()` is usually the easiest route. See [perlfaq4](#).

Floating-point numbers are only approximations to what a mathematician would call real numbers. There are infinitely more reals than floats, so some corners must be cut. For example:

```
printf "%.20g\n", 123456789123456789;
#           produces 123456789123456784
```

Testing for exact equality of floating-point equality or inequality is not a good idea. Here's a (relatively expensive) work-around to compare whether two floating-point numbers are equal to a particular number of decimal places. See Knuth, volume II, for a more robust treatment of this topic.

```
sub fp_equal {
    my ($X, $Y, $POINTS) = @_;
    my ($tX, $tY);
    $tX = sprintf("%.${POINTS}g", $X);
    $tY = sprintf("%.${POINTS}g", $Y);
    return $tX eq $tY;
}
```

The POSIX module (part of the standard perl distribution) implements `ceil()`, `floor()`, and other mathematical and trigonometric functions. The `Math::Complex` module (part of the standard perl distribution) defines mathematical functions that work on both the reals and the imaginary numbers. `Math::Complex` not as efficient as POSIX, but POSIX can't work with complex numbers.

Rounding in financial applications can have serious implications, and the rounding method used should be specified precisely. In these cases, it probably pays not to trust whichever system rounding is being used by Perl, but to instead implement the rounding function you need yourself.

Bigger Numbers

The standard `Math::BigInt` and `Math::BigFloat` modules provide variable-precision arithmetic and overloaded operators, although they're currently pretty slow. At the cost of some space and considerable speed, they avoid the normal pitfalls associated with limited-precision representations.

```
use Math::BigInt;
$x = Math::BigInt->new('123456789123456789');
print $x * $x;

# prints +15241578780673678515622620750190521
```

There are several modules that let you calculate with (bound only by memory and cpu-time) unlimited or fixed precision. There are also some non-standard modules that provide faster implementations via external C libraries.

Here is a short, but incomplete summary:

<code>Math::Fraction</code>	big, unlimited fractions like <code>9973 / 12967</code>
<code>Math::String</code>	treat string sequences like numbers
<code>Math::FixedPrecision</code>	calculate with a fixed precision
<code>Math::Currency</code>	for currency calculations
<code>Bit::Vector</code>	manipulate bit vectors fast (uses C)
<code>Math::BigIntFast</code>	<code>Bit::Vector</code> wrapper for big numbers

Math::Pari	provides access to the Pari C library
Math::BigInt	uses an external C library
Math::Cephes	uses external Cephes C library (no big numbers)
Math::Cephes::Fraction	fractions via the Cephes library
Math::GMP	another one using an external C library

Choose wisely.

NAME

perlopentut – tutorial on opening things in Perl

DESCRIPTION

Perl has two simple, built-in ways to open files: the shell way for convenience, and the C way for precision. The choice is yours.

Open à la shell

Perl's open function was designed to mimic the way command-line redirection in the shell works. Here are some basic examples from the shell:

```
$ myprogram file1 file2 file3
$ myprogram < inputfile
$ myprogram > outputfile
$ myprogram >> outputfile
$ myprogram | otherprogram
$ otherprogram | myprogram
```

And here are some more advanced examples:

```
$ otherprogram | myprogram f1 - f2
$ otherprogram 2>&1 | myprogram -
$ myprogram <&3
$ myprogram >&4
```

Programmers accustomed to constructs like those above can take comfort in learning that Perl directly supports these familiar constructs using virtually the same syntax as the shell.

Simple Opens

The open function takes two arguments: the first is a filehandle, and the second is a single string comprising both what to open and how to open it. open returns true when it works, and when it fails, returns a false value and sets the special variable \$! to reflect the system error. If the filehandle was previously opened, it will be implicitly closed first.

For example:

```
open(INFO, "datafile") || die("can't open datafile: $!");
open(INFO, "< datafile") || die("can't open datafile: $!");
open(RESULTS, "> runstats") || die("can't open runstats: $!");
open(LOG, ">> logfile ") || die("can't open logfile: $!");
```

If you prefer the low-punctuation version, you could write that this way:

```
open INFO, "< datafile" or die "can't open datafile: $!";
open RESULTS, "> runstats" or die "can't open runstats: $!";
open LOG, ">> logfile " or die "can't open logfile: $!";
```

A few things to notice. First, the leading less-than is optional. If omitted, Perl assumes that you want to open the file for reading.

The other important thing to notice is that, just as in the shell, any white space before or after the filename is ignored. This is good, because you wouldn't want these to do different things:

```
open INFO, "<datafile"
open INFO, "< datafile"
open INFO, "< datafile"
```

Ignoring surround whitespace also helps for when you read a filename in from a different file, and forget to trim it before opening:

```
$filename = <INFO>;          # oops, \n still there
```

```
open(EXTRA, "< $filename") || die "can't open $filename: $!";
```

This is not a bug, but a feature. Because `open` mimics the shell in its style of using redirection arrows to specify how to open the file, it also does so with respect to extra white space around the filename itself as well. For accessing files with naughty names, see ["Dispelling the Dweomer"](#).

Pipe Opens

In C, when you want to open a file using the standard I/O library, you use the `fopen` function, but when opening a pipe, you use the `popen` function. But in the shell, you just use a different redirection character. That's also the case for Perl. The `open` call remains the same—just its argument differs.

If the leading character is a pipe symbol, `open` starts up a new command and open a write-only filehandle leading into that command. This lets you write into that handle and have what you write show up on that command's standard input. For example:

```
open(PRINTER, "| lpr -Plp1")      || die "can't run lpr: $!";
print PRINTER "stuff\n";
close(PRINTER)                   || die "can't close lpr: $!";
```

If the trailing character is a pipe, you start up a new command and open a read-only filehandle leading out of that command. This lets whatever that command writes to its standard output show up on your handle for reading. For example:

```
open(NET, "netstat -i -n |")      || die "can't fun netstat: $!";
while (<NET>) { }                  # do something with input
close(NET)                        || die "can't close netstat: $!";
```

What happens if you try to open a pipe to or from a non-existent command? If possible, Perl will detect the failure and set `$!` as usual. But if the command contains special shell characters, such as `>` or `*`, called 'metacharacters', Perl does not execute the command directly. Instead, Perl runs the shell, which then tries to run the command. This means that it's the shell that gets the error indication. In such a case, the `open` call will only indicate failure if Perl can't even run the shell. See

[How can I capture STDERR from an external command? in perlfaq8](#) to see how to cope with this. There's also an explanation in [perlipc](#).

If you would like to open a bidirectional pipe, the `IPC::Open2` library will handle this for you. Check out [Bidirectional Communication with Another Process in perlipc](#)

The Minus File

Again following the lead of the standard shell utilities, Perl's `open` function treats a file whose name is a single minus, `"-"`, in a special way. If you open minus for reading, it really means to access the standard input. If you open minus for writing, it really means to access the standard output.

If minus can be used as the default input or default output, what happens if you open a pipe into or out of minus? What's the default command it would run? The same script as you're currently running! This is actually a stealth `fork` hidden inside an `open` call. See [Safe Pipe Opens in perlipc](#) for details.

Mixing Reads and Writes

It is possible to specify both read and write access. All you do is add a `+` symbol in front of the redirection. But as in the shell, using a less-than on a file never creates a new file; it only opens an existing one. On the other hand, using a greater-than always clobbers (truncates to zero length) an existing file, or creates a brand-new one if there isn't an old one. Adding a `+` for read-write doesn't affect whether it only works on existing files or always clobbers existing ones.

```
open(WTMP, "+< /usr/adm/wtmp")
|| die "can't open /usr/adm/wtmp: $!";

open(SCREEN, "+> /tmp/lkscreen")
|| die "can't open /tmp/lkscreen: $!";
```

```
open(LOGFILE, "+>> /tmp/applog"
    || die "can't open /tmp/applog: $!");
```

The first one won't create a new file, and the second one will always clobber an old one. The third one will create a new file if necessary and not clobber an old one, and it will allow you to read at any point in the file, but all writes will always go to the end. In short, the first case is substantially more common than the second and third cases, which are almost always wrong. (If you know C, the plus in Perl's `open` is historically derived from the one in C's `fopen(3S)`, which it ultimately calls.)

In fact, when it comes to updating a file, unless you're working on a binary file as in the WTMP case above, you probably don't want to use this approach for updating. Instead, Perl's `-i` flag comes to the rescue. The following command takes all the C, C++, or yacc source or header files and changes all their foo's to bar's, leaving the old version in the original file name with a ".orig" tacked on the end:

```
$ perl -i.orig -pe 's/\bfoo\b/bar/g' *.Cchy]
```

This is a short cut for some renaming games that are really the best way to update textfiles. See the second question in [perlfaq5](#) for more details.

Filters

One of the most common uses for `open` is one you never even notice. When you process the ARGV filehandle using `< <ARGV`, Perl actually does an implicit `open` on each file in `@ARGV`. Thus a program called like this:

```
$ myprogram file1 file2 file3
```

Can have all its files opened and processed one at a time using a construct no more complex than:

```
while (<>) {
    # do something with $_
}
```

If `@ARGV` is empty when the loop first begins, Perl pretends you've opened up minus, that is, the standard input. In fact, `$ARGV`, the currently open file during `< <ARGV` processing, is even set to "-" in these circumstances.

You are welcome to pre-process your `@ARGV` before starting the loop to make sure it's to your liking. One reason to do this might be to remove command options beginning with a minus. While you can always roll the simple ones by hand, the `Getopts` modules are good for this.

```
use Getopt::Std;

# -v, -D, -o ARG, sets $opt_v, $opt_D, $opt_o
getopts("vDo:");

# -v, -D, -o ARG, sets $args{v}, $args{D}, $args{o}
getopts("vDo:", \%args);
```

Or the standard `Getopt::Long` module to permit named arguments:

```
use Getopt::Long;
GetOptions( "verbose" => \$verbose,      # --verbose
            "Debug"   => \$debug,        # --Debug
            "output=s" => \$output );
# --output=somestring or --output somestring
```

Another reason for preprocessing arguments is to make an empty argument list default to all files:

```
@ARGV = glob("*") unless @ARGV;
```

You could even filter out all but plain, text files. This is a bit silent, of course, and you might prefer to mention them on the way.

```
@ARGV = grep { -f && -T } @ARGV;
```

If you're using the `-n` or `-p` command-line options, you should put changes to `@ARGV` in a `BEGIN{ }` block.

Remember that a normal `open` has special properties, in that it might call `fopen(3S)` or it might call `popen(3S)`, depending on what its argument looks like; that's why it's sometimes called "magic open". Here's an example:

```
$pwdinfo = `domainname` =~ /^(\(none\))?$/
           ? '< /etc/passwd'
           : `ypcat passwd |`;

open(PWD, $pwdinfo)
    or die "can't open $pwdinfo: $!";
```

This sort of thing also comes into play in filter processing. Because `< <ARGV` processing employs the normal, shell-style Perl `open`, it respects all the special things we've already seen:

```
$ myprogram f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

That program will read from the file *f1*, the process *cmd1*, standard input (*tmpfile* in this case), the *f2* file, the *cmd2* command, and finally the *f3* file.

Yes, this also means that if you have a file named `"-"` (and so on) in your directory, that they won't be processed as literal files by `open`. You'll need to pass them as `"./-"` much as you would for the *rm* program. Or you could use `sysopen` as described below.

One of the more interesting applications is to change files of a certain name into pipes. For example, to autoprocess gzipped or compressed files by decompressing them with *gzip*:

```
@ARGV = map { /^\. (gz|Z)$/ ? "gzip -dc $_|" : $_ } @ARGV;
```

Or, if you have the *GET* program installed from LWP, you can fetch URLs before processing them:

```
@ARGV = map { m#^\w+://# ? "GET $_|" : $_ } @ARGV;
```

It's not for nothing that this is called magic `< <ARGV`. Pretty nifty, eh?

Open à la C

If you want the convenience of the shell, then Perl's `open` is definitely the way to go. On the other hand, if you want finer precision than C's simplistic `fopen(3S)` provides, then you should look to Perl's `sysopen`, which is a direct hook into the `open(2)` system call. That does mean it's a bit more involved, but that's the price of precision.

`sysopen` takes 3 (or 4) arguments.

```
sysopen HANDLE, PATH, FLAGS, [MASK]
```

The `HANDLE` argument is a filehandle just as with `open`. The `PATH` is a literal path, one that doesn't pay attention to any greater-thans or less-thans or pipes or minuses, nor ignore white space. If it's there, it's part of the path. The `FLAGS` argument contains one or more values derived from the `Fcntl` module that have been or'd together using the bitwise `"|"` operator. The final argument, the `MASK`, is optional; if present, it is combined with the user's current `umask` for the creation mode of the file. You should usually omit this.

Although the traditional values of read-only, write-only, and read-write are 0, 1, and 2 respectively, this is known not to hold true on some systems. Instead, it's best to load in the appropriate constants first from the `Fcntl` module, which supplies the following standard flags:

<code>O_RDONLY</code>	Read only
<code>O_WRONLY</code>	Write only
<code>O_RDWR</code>	Read and write
<code>O_CREAT</code>	Create the file if it doesn't exist
<code>O_EXCL</code>	Fail if the file already exists

<code>O_APPEND</code>	Append to the file
<code>O_TRUNC</code>	Truncate the file
<code>O_NONBLOCK</code>	Non-blocking access

Less common flags that are sometimes available on some operating systems include `O_BINARY`, `O_TEXT`, `O_SHLOCK`, `O_EXLOCK`, `O_DEFER`, `O_SYNC`, `O_ASYNC`, `O_DSYNC`, `O_RSYNC`, `O_NOCTTY`, `O_NDELAY` and `O_LARGEFILE`. Consult your `open(2)` manpage or its local equivalent for details. (Note: starting from Perl release 5.6 the `O_LARGEFILE` flag, if available, is automatically added to the `sysopen()` flags because large files are the default.)

Here's how to use `sysopen` to emulate the simple `open` calls we had before. We'll omit the `|| die $!` checks for clarity, but make sure you always check the return values in real code. These aren't quite the same, since `open` will trim leading and trailing white space, but you'll get the idea:

To open a file for reading:

```
open(FH, "< $path");
sysopen(FH, $path, O_RDONLY);
```

To open a file for writing, creating a new file if needed or else truncating an old file:

```
open(FH, "> $path");
sysopen(FH, $path, O_WRONLY | O_TRUNC | O_CREAT);
```

To open a file for appending, creating one if necessary:

```
open(FH, ">> $path");
sysopen(FH, $path, O_WRONLY | O_APPEND | O_CREAT);
```

To open a file for update, where the file must already exist:

```
open(FH, "+< $path");
sysopen(FH, $path, O_RDWR);
```

And here are things you can do with `sysopen` that you cannot do with a regular `open`. As you see, it's just a matter of controlling the flags in the third argument.

To open a file for writing, creating a new file which must not previously exist:

```
sysopen(FH, $path, O_WRONLY | O_EXCL | O_CREAT);
```

To open a file for appending, where that file must already exist:

```
sysopen(FH, $path, O_WRONLY | O_APPEND);
```

To open a file for update, creating a new file if necessary:

```
sysopen(FH, $path, O_RDWR | O_CREAT);
```

To open a file for update, where that file must not already exist:

```
sysopen(FH, $path, O_RDWR | O_EXCL | O_CREAT);
```

To open a file without blocking, creating one if necessary:

```
sysopen(FH, $path, O_WRONLY | O_NONBLOCK | O_CREAT);
```

Permissions à la mode

If you omit the `MASK` argument to `sysopen`, Perl uses the octal value 0666. The normal `MASK` to use for executables and directories should be 0777, and for anything else, 0666.

Why so permissive? Well, it isn't really. The `MASK` will be modified by your process's current `umask`. A `umask` is a number representing *disabled* permissions bits; that is, bits that will not be turned on in the created files' permissions field.

For example, if your `umask` were 027, then the 020 part would disable the group from writing, and the 007

part would disable others from reading, writing, or executing. Under these conditions, passing `sysopen 0666` would create a file with mode 0640, since `0666 &~ 027` is 0640.

You should seldom use the `MASK` argument to `sysopen()`. That takes away the user's freedom to choose what permission new files will have. Denying choice is almost always a bad thing. One exception would be for cases where sensitive or private data is being stored, such as with mail folders, cookie files, and internal temporary files.

Obscure Open Tricks

Re-Opening Files (dups)

Sometimes you already have a filehandle open, and want to make another handle that's a duplicate of the first one. In the shell, we place an ampersand in front of a file descriptor number when doing redirections. For example, `< 2&1` makes descriptor 2 (that's `STDERR` in Perl) be redirected into descriptor 1 (which is usually Perl's `STDOUT`). The same is essentially true in Perl: a filename that begins with an ampersand is treated instead as a file descriptor if a number, or as a filehandle if a string.

```
open(SAVEOUT, ">&SAVEERR") || die "couldn't dup SAVEERR: $!";
open(MHCONTEXT, "<&4")      || die "couldn't dup fd4: $!";
```

That means that if a function is expecting a filename, but you don't want to give it a filename because you already have the file open, you can just pass the filehandle with a leading ampersand. It's best to use a fully qualified handle though, just in case the function happens to be in a different package:

```
somefunction("&main::LOGFILE");
```

This way if `somefunction()` is planning on opening its argument, it can just use the already opened handle. This differs from passing a handle, because with a handle, you don't open the file. Here you have something you can pass to open.

If you have one of those tricky, newfangled I/O objects that the C++ folks are raving about, then this doesn't work because those aren't a proper filehandle in the native Perl sense. You'll have to use `fileno()` to pull out the proper descriptor number, assuming you can:

```
use IO::Socket;
$handle = IO::Socket::INET->new("www.perl.com:80");
$fd = $handle->fileno;
somefunction("&$fd"); # not an indirect function call
```

It can be easier (and certainly will be faster) just to use real filehandles though:

```
use IO::Socket;
local *REMOTE = IO::Socket::INET->new("www.perl.com:80");
die "can't connect" unless defined(fileno(REMOTE));
somefunction("&main::REMOTE");
```

If the filehandle or descriptor number is preceded not just with a simple "&" but rather with a "&=" combination, then Perl will not create a completely new descriptor opened to the same place using the `dup(2)` system call. Instead, it will just make something of an alias to the existing one using the `fdopen(3S)` library call. This is slightly more parsimonious of systems resources, although this is less a concern these days. Here's an example of that:

```
$fd = $ENV{"MHCONTEXTFD"};
open(MHCONTEXT, "<&=$fd") or die "couldn't fdopen $fd: $!";
```

If you're using magic `< <ARGV`, you could even pass in as a command line argument in `@ARGV` something like `"<&=$MHCONTEXTFD"`, but we've never seen anyone actually do this.

Dispelling the Dweomer

Perl is more of a DWIMmer language than something like Java—where DWIM is an acronym for "do what I mean". But this principle sometimes leads to more hidden magic than one knows what to do with. In this way, Perl is also filled with *dweomer*, an obscure word meaning an enchantment. Sometimes, Perl's

DWIMmer is just too much like dweomer for comfort.

If magic `open` is a bit too magical for you, you don't have to turn to `sysopen`. To open a file with arbitrary weird characters in it, it's necessary to protect any leading and trailing whitespace. Leading whitespace is protected by inserting a `"/` in front of a filename that starts with whitespace. Trailing whitespace is protected by appending an ASCII NUL byte (`"\0"`) at the end of the string.

```
$file =~ s#^\s)#./$1#;
open(FH, "< $file\0") || die "can't open $file: $!";
```

This assumes, of course, that your system considers dot the current working directory, slash the directory separator, and disallows ASCII NULs within a valid filename. Most systems follow these conventions, including all POSIX systems as well as proprietary Microsoft systems. The only vaguely popular system that doesn't work this way is the proprietary Macintosh system, which uses a colon where the rest of us use a slash. Maybe `sysopen` isn't such a bad idea after all.

If you want to use `< <ARGV` processing in a totally boring and non-magical way, you could do this first:

```
# "Sam sat on the ground and put his head in his hands.
# 'I wish I had never come here, and I don't want to see
# no more magic,' he said, and fell silent."
for (@ARGV) {
    s#^\s)#./$1#;
    $_ .= "\0";
}
while (<>) {
    # now process $_
}
```

But be warned that users will not appreciate being unable to use `"-"` to mean standard input, per the standard convention.

Paths as Opens

You've probably noticed how Perl's `warn` and `die` functions can produce messages like:

```
Some warning at scriptname line 29, <FH> line 7.
```

That's because you opened a filehandle `FH`, and had read in seven records from it. But what was the name of the file, not the handle?

If you aren't running with `strict refs`, or if you've turn them off temporarily, then all you have to do is this:

```
open($path, "< $path") || die "can't open $path: $!";
while (<$path>) {
    # whatever
}
```

Since you're using the pathname of the file as its handle, you'll get warnings more like

```
Some warning at scriptname line 29, </etc/motd> line 7.
```

Single Argument Open

Remember how we said that Perl's `open` took two arguments? That was a passive prevarication. You see, it can also take just one argument. If and only if the variable is a global variable, not a lexical, you can pass `open` just one argument, the filehandle, and it will get the path from the global scalar variable of the same name.

```
$FILE = "/etc/motd";
open FILE or die "can't open $FILE: $!";
while (<FILE>) {
    # whatever
}
```

```
}
```

Why is this here? Someone has to cater to the hysterical porpoises. It's something that's been in Perl since the very beginning, if not before.

Playing with STDIN and STDOUT

One clever move with STDOUT is to explicitly close it when you're done with the program.

```
END { close(STDOUT) || die "can't close stdout: $!" }
```

If you don't do this, and your program fills up the disk partition due to a command line redirection, it won't report the error exit with a failure status.

You don't have to accept the STDIN and STDOUT you were given. You are welcome to reopen them if you'd like.

```
open(STDIN, "< datafile")
|| die "can't open datafile: $!";

open(STDOUT, "> output")
|| die "can't open output: $!";
```

And then these can be read directly or passed on to subprocesses. This makes it look as though the program were initially invoked with those redirections from the command line.

It's probably more interesting to connect these to pipes. For example:

```
$pager = $ENV{PAGER} || "(less || more)";
open(STDOUT, "| $pager")
|| die "can't fork a pager: $!";
```

This makes it appear as though your program were called with its stdout already piped into your pager. You can also use this kind of thing in conjunction with an implicit fork to yourself. You might do this if you would rather handle the post processing in your own program, just in a different process:

```
head(100);
while (<>) {
    print;
}

sub head {
    my $lines = shift || 20;
    return unless $pid = open(STDOUT, "|-");
    die "cannot fork: $!" unless defined $pid;
    while (<STDIN>) {
        print;
        last if --$lines < 0;
    }
    exit;
}
```

This technique can be applied to repeatedly push as many filters on your output stream as you wish.

Other I/O Issues

These topics aren't really arguments related to `open` or `sysopen`, but they do affect what you do with your open files.

Opening Non-File Files

When is a file not a file? Well, you could say when it exists but isn't a plain file. We'll check whether it's a symbolic link first, just in case.

```
if (-l $file || ! -f _) {
```

```
    print "$file is not a plain file\n";
}
```

What other kinds of files are there than, well, files? Directories, symbolic links, named pipes, Unix-domain sockets, and block and character devices. Those are all files, too—just not *plain* files. This isn't the same issue as being a text file. Not all text files are plain files. Not all plain files are textfiles. That's why there are separate `-f` and `-T` file tests.

To open a directory, you should use the `opendir` function, then process it with `readdir`, carefully restoring the directory name if necessary:

```
opendir(DIR, $dirname) or die "can't opendir $dirname: $!";
while (defined($file = readdir(DIR))) {
    # do something with "$dirname/$file"
}
closedir(DIR);
```

If you want to process directories recursively, it's better to use the `File::Find` module. For example, this prints out all files recursively, add adds a slash to their names if the file is a directory.

```
@ARGV = qw(.) unless @ARGV;
use File::Find;
find sub { print $File::Find::name, -d && '/' , "\n" }, @ARGV;
```

This finds all bogus symbolic links beneath a particular directory:

```
find sub { print "$File::Find::name\n" if -l && !-e }, $dir;
```

As you see, with symbolic links, you can just pretend that it is what it points to. Or, if you want to know *what* it points to, then `readlink` is called for:

```
if (-l $file) {
    if (defined($whither = readlink($file))) {
        print "$file points to $whither\n";
    } else {
        print "$file points nowhere: $!\n";
    }
}
```

Named pipes are a different matter. You pretend they're regular files, but their opens will normally block until there is both a reader and a writer. You can read more about them in [Named Pipes in perlipc](#). Unix-domain sockets are rather different beasts as well; they're described in [Unix-Domain TCP Clients and Servers in perlipc](#).

When it comes to opening devices, it can be easy and it can tricky. We'll assume that if you're opening up a block device, you know what you're doing. The character devices are more interesting. These are typically used for modems, mice, and some kinds of printers. This is described in [How do I read and write the serial port? in perlfaq8](#) It's often enough to open them carefully:

```
sysopen(TTYIN, "/dev/ttyS1", O_RDWR | O_NDELAY | O_NOCTTY)
    # (O_NOCTTY no longer needed on POSIX systems)
or die "can't open /dev/ttyS1: $!";
open(TTYOUT, "+>&TTYIN")
or die "can't dup TTYIN: $!";

$ofh = select(TTYOUT); $| = 1; select($ofh);

print TTYOUT "+++at\015";
$answer = <TTYIN>;
```

With descriptors that you haven't opened using `sysopen`, such as a socket, you can set them to be non-blocking using `fcntl`:

```
use Fcntl;
fcntl(Connection, F_SETFL, O_NONBLOCK)
    or die "can't set non blocking: $!";
```

Rather than losing yourself in a morass of twisting, turning `ioctl`s, all dissimilar, if you're going to manipulate ttys, it's best to make calls out to the `stty(1)` program if you have it, or else use the portable POSIX interface. To figure this all out, you'll need to read the `termios(3)` manpage, which describes the POSIX interface to tty devices, and then *POSIX*, which describes Perl's interface to POSIX. There are also some high-level modules on CPAN that can help you with these games. Check out `Term::ReadKey` and `Term::ReadLine`.

What else can you open? To open a connection using sockets, you won't use one of Perl's two open functions. See *Sockets: Client/Server Communication in `perlipc`* for that. Here's an example. Once you have it, you can use FH as a bidirectional filehandle.

```
use IO::Socket;
local *FH = IO::Socket::INET->new("www.perl.com:80");
```

For opening up a URL, the LWP modules from CPAN are just what the doctor ordered. There's no filehandle interface, but it's still easy to get the contents of a document:

```
use LWP::Simple;
$doc = get('http://www.linpro.no/lwp/');
```

Binary Files

On certain legacy systems with what could charitably be called terminally convoluted (some would say broken) I/O models, a file isn't a file—at least, not with respect to the C standard I/O library. On these old systems whose libraries (but not kernels) distinguish between text and binary streams, to get files to behave properly you'll have to bend over backwards to avoid nasty problems. On such infelicitous systems, sockets and pipes are already opened in binary mode, and there is currently no way to turn that off. With files, you have more options.

Another option is to use the `binmode` function on the appropriate handles before doing regular I/O on them:

```
binmode(STDIN);
binmode(STDOUT);
while (<STDIN>) { print }
```

Passing `sysopen` a non-standard flag option will also open the file in binary mode on those systems that support it. This is the equivalent of opening the file normally, then calling `binmode` on the handle.

```
sysopen(BINDAT, "records.data", O_RDWR | O_BINARY)
    || die "can't open records.data: $!";
```

Now you can use `read` and `print` on that handle without worrying about the system non-standard I/O library breaking your data. It's not a pretty picture, but then, legacy systems seldom are. CP/M will be with us until the end of days, and after.

On systems with exotic I/O systems, it turns out that, astonishingly enough, even unbuffered I/O using `sysread` and `syswrite` might do sneaky data mutilation behind your back.

```
while (sysread(WHENCE, $buf, 1024)) {
    syswrite(WHITHER, $buf, length($buf));
}
```

Depending on the vicissitudes of your runtime system, even these calls may need `binmode` or `O_BINARY` first. Systems known to be free of such difficulties include Unix, the Mac OS, Plan 9, and Inferno.

File Locking

In a multitasking environment, you may need to be careful not to collide with other processes who want to do I/O on the same files as others are working on. You'll often need shared or exclusive locks on files for reading and writing respectively. You might just pretend that only exclusive locks exist.

Never use the existence of a file `-e $file` as a locking indication, because there is a race condition between the test for the existence of the file and its creation. Atomicity is critical.

Perl's most portable locking interface is via the `flock` function, whose simplicity is emulated on systems that don't directly support it, such as SysV or WindowsNT. The underlying semantics may affect how it all works, so you should learn how `flock` is implemented on your system's port of Perl.

File locking *does not* lock out another process that would like to do I/O. A file lock only locks out others trying to get a lock, not processes trying to do I/O. Because locks are advisory, if one process uses locking and another doesn't, all bets are off.

By default, the `flock` call will block until a lock is granted. A request for a shared lock will be granted as soon as there is no exclusive locker. A request for an exclusive lock will be granted as soon as there is no locker of any kind. Locks are on file descriptors, not file names. You can't lock a file until you open it, and you can't hold on to a lock once the file has been closed.

Here's how to get a blocking shared lock on a file, typically used for reading:

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename") or die "can't open filename: $!";
flock(FH, LOCK_SH)      or die "can't lock filename: $!";
# now read from FH
```

You can get a non-blocking lock by using `LOCK_NB`.

```
flock(FH, LOCK_SH | LOCK_NB)
    or die "can't lock filename: $!";
```

This can be useful for producing more user-friendly behaviour by warning if you're going to be blocking:

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename") or die "can't open filename: $!";
unless (flock(FH, LOCK_SH | LOCK_NB)) {
    $| = 1;
    print "Waiting for lock...";
    flock(FH, LOCK_SH) or die "can't lock filename: $!";
    print "got it.\n"
}
# now read from FH
```

To get an exclusive lock, typically used for writing, you have to be careful. We `sysopen` the file so it can be locked before it gets emptied. You can get a nonblocking version using `LOCK_EX | LOCK_NB`.

```
use 5.004;
use Fcntl qw(:DEFAULT :flock);
sysopen(FH, "filename", O_WRONLY | O_CREAT)
    or die "can't open filename: $!";
flock(FH, LOCK_EX)
    or die "can't lock filename: $!";
truncate(FH, 0)
    or die "can't truncate filename: $!";
# now write to FH
```

Finally, due to the uncounted millions who cannot be dissuaded from wasting cycles on useless vanity devices called hit counters, here's how to increment a number in a file safely:

```
use Fcntl qw(:DEFAULT :flock);

sysopen(FH, "numfile", O_RDWR | O_CREAT)
    or die "can't open numfile: $!";
# autoflush FH
$ofh = select(FH); $| = 1; select ($ofh);
flock(FH, LOCK_EX)
    or die "can't write-lock numfile: $!";

$num = <FH> || 0;
seek(FH, 0, 0)
    or die "can't rewind numfile : $!";
print FH $num+1, "\n"
    or die "can't write numfile: $!";

truncate(FH, tell(FH))
    or die "can't truncate numfile: $!";
close(FH)
    or die "can't close numfile: $!";
```

IO Layers

In Perl 5.8.0 a new I/O framework called "PerlIO" was introduced. This is a new "plumbing" for all the I/O happening in Perl; for the most part everything will work just as it did, but PerlIO brought in also some new features, like the capability of think of I/O as "layers". One I/O layer may in addition to just moving the data also do transformations on the data. Such transformations may include compression and decompression, encryption and decryption, and transforming between various character encodings.

Full discussion about the features of PerlIO is out of scope for this tutorial, but here is how to recognize the layers being used:

- The three-(or more)-argument form of `open()` is being used and the second argument contains something else in addition to the usual `< '<' , < '>' , < '>' , < '|'` and their variants, for example:

```
open(my $fh, "<:utf8", $fn);
```

- The two-argument form of `binmode<open()` is being used, for example

```
binmode($fh, ":encoding(utf16)");
```

For more detailed discussion about PerlIO see [perlio](#); for more detailed discussion about Unicode and I/O see [perluniintro](#).

SEE ALSO

The `open` and `sysopen` function in `perlfunc(1)`; the standard `open(2)`, `dup(2)`, `fopen(3)`, and `fdopen(3)` manpages; the POSIX documentation.

AUTHOR and COPYRIGHT

Copyright 1998 Tom Christiansen.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in these files are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

HISTORY

First release: Sat Jan 9 08:09:11 MST 1999

NAME

perlothrtut – old tutorial on threads in Perl

DESCRIPTION

WARNING: This tutorial describes the old-style thread model that was introduced in release 5.005. This model is now deprecated, and will be removed, probably in version 5.10. The interfaces described here were considered experimental, and are likely to be buggy.

For information about the new interpreter threads ("ithreads") model, see the *perlthrtut* tutorial, and the *threads* and *threads::shared* modules.

You are strongly encouraged to migrate any existing threads code to the new model as soon as possible.

What Is A Thread Anyway?

A thread is a flow of control through a program with a single execution point.

Sounds an awful lot like a process, doesn't it? Well, it should. Threads are one of the pieces of a process. Every process has at least one thread and, up until now, every process running Perl had only one thread. With 5.005, though, you can create extra threads. We're going to show you how, when, and why.

Threaded Program Models

There are three basic ways that you can structure a threaded program. Which model you choose depends on what you need your program to do. For many non-trivial threaded programs you'll need to choose different models for different pieces of your program.

Boss/Worker

The boss/worker model usually has one 'boss' thread and one or more 'worker' threads. The boss thread gathers or generates tasks that need to be done, then parcels those tasks out to the appropriate worker thread.

This model is common in GUI and server programs, where a main thread waits for some event and then passes that event to the appropriate worker threads for processing. Once the event has been passed on, the boss thread goes back to waiting for another event.

The boss thread does relatively little work. While tasks aren't necessarily performed faster than with any other method, it tends to have the best user-response times.

Work Crew

In the work crew model, several threads are created that do essentially the same thing to different pieces of data. It closely mirrors classical parallel processing and vector processors, where a large array of processors do the exact same thing to many pieces of data.

This model is particularly useful if the system running the program will distribute multiple threads across different processors. It can also be useful in ray tracing or rendering engines, where the individual threads can pass on interim results to give the user visual feedback.

Pipeline

The pipeline model divides up a task into a series of steps, and passes the results of one step on to the thread processing the next. Each thread does one thing to each piece of data and passes the results to the next thread in line.

This model makes the most sense if you have multiple processors so two or more threads will be executing in parallel, though it can often make sense in other contexts as well. It tends to keep the individual tasks small and simple, as well as allowing some parts of the pipeline to block (on I/O or system calls, for example) while other parts keep going. If you're running different parts of the pipeline on different processors you may also take advantage of the caches on each processor.

This model is also handy for a form of recursive programming where, rather than having a subroutine call itself, it instead creates another thread. Prime and Fibonacci generators both map well to this form of the pipeline model. (A version of a prime number generator is presented later on.)

Native threads

There are several different ways to implement threads on a system. How threads are implemented depends both on the vendor and, in some cases, the version of the operating system. Often the first implementation will be relatively simple, but later versions of the OS will be more sophisticated.

While the information in this section is useful, it's not necessary, so you can skip it if you don't feel up to it.

There are three basic categories of threads—user-mode threads, kernel threads, and multiprocessor kernel threads.

User-mode threads are threads that live entirely within a program and its libraries. In this model, the OS knows nothing about threads. As far as it's concerned, your process is just a process.

This is the easiest way to implement threads, and the way most OSes start. The big disadvantage is that, since the OS knows nothing about threads, if one thread blocks they all do. Typical blocking activities include most system calls, most I/O, and things like `sleep()`.

Kernel threads are the next step in thread evolution. The OS knows about kernel threads, and makes allowances for them. The main difference between a kernel thread and a user-mode thread is blocking. With kernel threads, things that block a single thread don't block other threads. This is not the case with user-mode threads, where the kernel blocks at the process level and not the thread level.

This is a big step forward, and can give a threaded program quite a performance boost over non-threaded programs. Threads that block performing I/O, for example, won't block threads that are doing other things. Each process still has only one thread running at once, though, regardless of how many CPUs a system might have.

Since kernel threading can interrupt a thread at any time, they will uncover some of the implicit locking assumptions you may make in your program. For example, something as simple as `$a = $a + 2` can behave unpredictably with kernel threads if `$a` is visible to other threads, as another thread may have changed `$a` between the time it was fetched on the right hand side and the time the new value is stored.

Multiprocessor Kernel Threads are the final step in thread support. With multiprocessor kernel threads on a machine with multiple CPUs, the OS may schedule two or more threads to run simultaneously on different CPUs.

This can give a serious performance boost to your threaded program, since more than one thread will be executing at the same time. As a tradeoff, though, any of those nagging synchronization issues that might not have shown with basic kernel threads will appear with a vengeance.

In addition to the different levels of OS involvement in threads, different OSes (and different thread implementations for a particular OS) allocate CPU cycles to threads in different ways.

Cooperative multitasking systems have running threads give up control if one of two things happen. If a thread calls a yield function, it gives up control. It also gives up control if the thread does something that would cause it to block, such as perform I/O. In a cooperative multitasking implementation, one thread can starve all the others for CPU time if it so chooses.

Preemptive multitasking systems interrupt threads at regular intervals while the system decides which thread should run next. In a preemptive multitasking system, one thread usually won't monopolize the CPU.

On some systems, there can be cooperative and preemptive threads running simultaneously. (Threads running with realtime priorities often behave cooperatively, for example, while threads running at normal priorities behave preemptively.)

What kind of threads are perl threads?

If you have experience with other thread implementations, you might find that things aren't quite what you expect. It's very important to remember when dealing with Perl threads that Perl Threads Are Not X Threads, for all values of X. They aren't POSIX threads, or DecThreads, or Java's Green threads, or Win32 threads. There are similarities, and the broad concepts are the same, but if you start looking for implementation details you're going to be either disappointed or confused. Possibly both.

This is not to say that Perl threads are completely different from everything that's ever come before—they're not. Perl's threading model owes a lot to other thread models, especially POSIX. Just as Perl is not C, though, Perl threads are not POSIX threads. So if you find yourself looking for mutexes, or thread priorities, it's time to step back a bit and think about what you want to do and how Perl can do it.

Threadsafe Modules

The addition of threads has changed Perl's internals substantially. There are implications for people who write modules—especially modules with XS code or external libraries. While most modules won't encounter any problems, modules that aren't explicitly tagged as thread-safe should be tested before being used in production code.

Not all modules that you might use are thread-safe, and you should always assume a module is unsafe unless the documentation says otherwise. This includes modules that are distributed as part of the core. Threads are a beta feature, and even some of the standard modules aren't thread-safe.

If you're using a module that's not thread-safe for some reason, you can protect yourself by using semaphores and lots of programming discipline to control access to the module. Semaphores are covered later in the article. Perl Threads Are Different

Thread Basics

The core Thread module provides the basic functions you need to write threaded programs. In the following sections we'll cover the basics, showing you what you need to do to create a threaded program. After that, we'll go over some of the features of the Thread module that make threaded programming easier.

Basic Thread Support

Thread support is a Perl compile-time option—it's something that's turned on or off when Perl is built at your site, rather than when your programs are compiled. If your Perl wasn't compiled with thread support enabled, then any attempt to use threads will fail.

Remember that the threading support in 5.005 is in beta release, and should be treated as such. You should expect that it may not function entirely properly, and the thread interface may well change some before it is a fully supported, production release. The beta version shouldn't be used for mission-critical projects. Having said that, threaded Perl is pretty nifty, and worth a look.

Your programs can use the Config module to check whether threads are enabled. If your program can't run without them, you can say something like:

```
$Config{usethreads} or die "Recompile Perl with threads to run this program.";
```

A possibly-threaded program using a possibly-threaded module might have code like this:

```
use Config;
use MyMod;

if ($Config{usethreads}) {
    # We have threads
    require MyMod_threaded;
    import MyMod_threaded;
} else {
    require MyMod_unthreaded;
    import MyMod_unthreaded;
}
```

Since code that runs both with and without threads is usually pretty messy, it's best to isolate the thread-specific code in its own module. In our example above, that's what MyMod_threaded is, and it's only imported if we're running on a threaded Perl.

Creating Threads

The Thread package provides the tools you need to create new threads. Like any other module, you need to tell Perl you want to use it; use Thread imports all the pieces you need to create basic threads.

The simplest, straightforward way to create a thread is with `new()`:

```
use Thread;

$thr = new Thread \&sub1;

sub sub1 {
    print "In the thread\n";
}
```

The `new()` method takes a reference to a subroutine and creates a new thread, which starts executing in the referenced subroutine. Control then passes both to the subroutine and the caller.

If you need to, your program can pass parameters to the subroutine as part of the thread startup. Just include the list of parameters as part of the `Thread::new` call, like this:

```
use Thread;
$Param3 = "foo";
$thr = new Thread \&sub1, "Param 1", "Param 2", $Param3;
$thr = new Thread \&sub1, @ParamList;
$thr = new Thread \&sub1, qw(Param1 Param2 $Param3);

sub sub1 {
    my @InboundParameters = @_;
    print "In the thread\n";
    print "got parameters >", join("<>", @InboundParameters), "<\n";
}
```

The subroutine runs like a normal Perl subroutine, and the call to `new Thread` returns whatever the subroutine returns.

The last example illustrates another feature of threads. You can spawn off several threads using the same subroutine. Each thread executes the same subroutine, but in a separate thread with a separate environment and potentially separate arguments.

The other way to spawn a new thread is with `async()`, which is a way to spin off a chunk of code like `eval()`, but into its own thread:

```
use Thread qw(async);

$LineCount = 0;

$thr = async {
    while(<>) {$LineCount++}
    print "Got $LineCount lines\n";
};

print "Waiting for the linecount to end\n";
$thr->join;
print "All done\n";
```

You'll notice we did a `use Thread qw(async)` in that example. `async` is not exported by default, so if you want it, you'll either need to import it before you use it or fully qualify it as `Thread::async`. You'll also note that there's a semicolon after the closing brace. That's because `async()` treats the following block as an anonymous subroutine, so the semicolon is necessary.

Like `eval()`, the code executes in the same context as it would if it weren't spun off. Since both the code inside and after the `async` start executing, you need to be careful with any shared resources. Locking and other synchronization techniques are covered later.

Giving up control

There are times when you may find it useful to have a thread explicitly give up the CPU to another thread. Your threading package might not support preemptive multitasking for threads, for example, or you may be doing something compute-intensive and want to make sure that the user-interface thread gets called frequently. Regardless, there are times that you might want a thread to give up the processor.

Perl's threading package provides the `yield()` function that does this. `yield()` is pretty straightforward, and works like this:

```
use Thread qw(yield async);
async {
    my $foo = 50;
    while ($foo-- > 0) { print "first async\n" }
    yield;
    $foo = 50;
    while ($foo-- > 0) { print "first async\n" }
};
async {
    my $foo = 50;
    while ($foo-- > 0) { print "second async\n" }
    yield;
    $foo = 50;
    while ($foo-- > 0) { print "second async\n" }
};
```

Waiting For A Thread To Exit

Since threads are also subroutines, they can return values. To wait for a thread to exit and extract any scalars it might return, you can use the `join()` method.

```
use Thread;
$thr = new Thread \&sub1;

@ReturnData = $thr->join;
print "Thread returned @ReturnData";

sub sub1 { return "Fifty-six", "foo", 2; }
```

In the example above, the `join()` method returns as soon as the thread ends. In addition to waiting for a thread to finish and gathering up any values that the thread might have returned, `join()` also performs any OS cleanup necessary for the thread. That cleanup might be important, especially for long-running programs that spawn lots of threads. If you don't want the return values and don't want to wait for the thread to finish, you should call the `detach()` method instead. `detach()` is covered later in the article.

Errors In Threads

So what happens when an error occurs in a thread? Any errors that could be caught with `eval()` are postponed until the thread is joined. If your program never joins, the errors appear when your program exits.

Errors deferred until a `join()` can be caught with `eval()`:

```
use Thread qw(async);
$thr = async { $b = 3/0; }; # Divide by zero error
$foo = eval { $thr->join };
if ($?) {
    print "died with error $@\n";
} else {
    print "Hey, why aren't you dead?\n";
}
```

`eval()` passes any results from the joined thread back unmodified, so if you want the return value of the thread, this is your only chance to get them.

Ignoring A Thread

`join()` does three things: it waits for a thread to exit, cleans up after it, and returns any data the thread may have produced. But what if you're not interested in the thread's return values, and you don't really care when the thread finishes? All you want is for the thread to get cleaned up after when it's done.

In this case, you use the `detach()` method. Once a thread is detached, it'll run until it's finished, then Perl will clean up after it automatically.

```
use Thread;
$thr = new Thread \&sub1; # Spawn the thread

$thr->detach; # Now we officially don't care any more

sub sub1 {
    $a = 0;
    while (1) {
        $a++;
        print "\$a is $a\n";
        sleep 1;
    }
}
```

Once a thread is detached, it may not be joined, and any output that it might have produced (if it was done and waiting for a join) is lost.

Threads And Data

Now that we've covered the basics of threads, it's time for our next topic: data. Threading introduces a couple of complications to data access that non-threaded programs never need to worry about.

Shared And Unshared Data

The single most important thing to remember when using threads is that all threads potentially have access to all the data anywhere in your program. While this is true with a nonthreaded Perl program as well, it's especially important to remember with a threaded program, since more than one thread can be accessing this data at once.

Perl's scoping rules don't change because you're using threads. If a subroutine (or block, in the case of `async()`) could see a variable if you weren't running with threads, it can see it if you are. This is especially important for the subroutines that create, and makes my variables even more important. Remember—if your variables aren't lexically scoped (declared with `my`) you're probably sharing them between threads.

Thread Pitfall: Races

While threads bring a new set of useful tools, they also bring a number of pitfalls. One pitfall is the race condition:

```
use Thread;
$a = 1;
$thr1 = Thread->new(\&sub1);
$thr2 = Thread->new(\&sub2);

sleep 10;
print "$a\n";

sub sub1 { $foo = $a; $a = $foo + 1; }
sub sub2 { $bar = $a; $a = $bar + 1; }
```

What do you think `$a` will be? The answer, unfortunately, is "it depends." Both `sub1()` and `sub2()` access the global variable `$a`, once to read and once to write. Depending on factors ranging from your

thread implementation's scheduling algorithm to the phase of the moon, \$a can be 2 or 3.

Race conditions are caused by unsynchronized access to shared data. Without explicit synchronization, there's no way to be sure that nothing has happened to the shared data between the time you access it and the time you update it. Even this simple code fragment has the possibility of error:

```
use Thread qw(async);
$a = 2;
async{ $b = $a; $a = $b + 1; };
async{ $c = $a; $a = $c + 1; };
```

Two threads both access \$a. Each thread can potentially be interrupted at any point, or be executed in any order. At the end, \$a could be 3 or 4, and both \$b and \$c could be 2 or 3.

Whenever your program accesses data or resources that can be accessed by other threads, you must take steps to coordinate access or risk data corruption and race conditions.

Controlling access: lock()

The `lock()` function takes a variable (or subroutine, but we'll get to that later) and puts a lock on it. No other thread may lock the variable until the locking thread exits the innermost block containing the lock. Using `lock()` is straightforward:

```
use Thread qw(async);
$a = 4;
$thr1 = async {
    $foo = 12;
    {
        lock ($a); # Block until we get access to $a
        $b = $a;
        $a = $b * $foo;
    }
    print "\$foo was $foo\n";
};
$thr2 = async {
    $bar = 7;
    {
        lock ($a); # Block until we can get access to $a
        $c = $a;
        $a = $c * $bar;
    }
    print "\$bar was $bar\n";
};
$thr1->join;
$thr2->join;
print "\$a is $a\n";
```

`lock()` blocks the thread until the variable being locked is available. When `lock()` returns, your thread can be sure that no other thread can lock that variable until the innermost block containing the lock exits.

It's important to note that locks don't prevent access to the variable in question, only lock attempts. This is in keeping with Perl's longstanding tradition of courteous programming, and the advisory file locking that `flock()` gives you. Locked subroutines behave differently, however. We'll cover that later in the article.

You may lock arrays and hashes as well as scalars. Locking an array, though, will not block subsequent locks on array elements, just lock attempts on the array itself.

Finally, locks are recursive, which means it's okay for a thread to lock a variable more than once. The lock will last until the outermost `lock()` on the variable goes out of scope.

Thread Pitfall: Deadlocks

Locks are a handy tool to synchronize access to data. Using them properly is the key to safe shared data. Unfortunately, locks aren't without their dangers. Consider the following code:

```
use Thread qw(async yield);
$a = 4;
$b = "foo";
async {
    lock($a);
    yield;
    sleep 20;
    lock ($b);
};
async {
    lock($b);
    yield;
    sleep 20;
    lock ($a);
};
```

This program will probably hang until you kill it. The only way it won't hang is if one of the two `async()` routines acquires both locks first. A guaranteed-to-hang version is more complicated, but the principle is the same.

The first thread spawned by `async()` will grab a lock on `$a` then, a second or two later, try to grab a lock on `$b`. Meanwhile, the second thread grabs a lock on `$b`, then later tries to grab a lock on `$a`. The second lock attempt for both threads will block, each waiting for the other to release its lock.

This condition is called a deadlock, and it occurs whenever two or more threads are trying to get locks on resources that the others own. Each thread will block, waiting for the other to release a lock on a resource. That never happens, though, since the thread with the resource is itself waiting for a lock to be released.

There are a number of ways to handle this sort of problem. The best way is to always have all threads acquire locks in the exact same order. If, for example, you lock variables `$a`, `$b`, and `$c`, always lock `$a` before `$b`, and `$b` before `$c`. It's also best to hold on to locks for as short a period of time to minimize the risks of deadlock.

Queues: Passing Data Around

A queue is a special thread-safe object that lets you put data in one end and take it out the other without having to worry about synchronization issues. They're pretty straightforward, and look like this:

```
use Thread qw(async);
use Thread::Queue;

my $DataQueue = new Thread::Queue;
$thr = async {
    while ($DataElement = $DataQueue->dequeue) {
        print "Popped $DataElement off the queue\n";
    }
};

$DataQueue->enqueue(12);
$DataQueue->enqueue("A", "B", "C");
$DataQueue->enqueue(\$thr);
sleep 10;
$DataQueue->enqueue(undef);
```

You create the queue with `new Thread::Queue`. Then you can add lists of scalars onto the end with `enqueue()`, and pop scalars off the front of it with `dequeue()`. A queue has no fixed size, and can grow

as needed to hold everything pushed on to it.

If a queue is empty, `dequeue()` blocks until another thread enqueues something. This makes queues ideal for event loops and other communications between threads.

Threads And Code

In addition to providing thread-safe access to data via locks and queues, threaded Perl also provides general-purpose semaphores for coarser synchronization than locks provide and thread-safe access to entire subroutines.

Semaphores: Synchronizing Data Access

Semaphores are a kind of generic locking mechanism. Unlike lock, which gets a lock on a particular scalar, Perl doesn't associate any particular thing with a semaphore so you can use them to control access to anything you like. In addition, semaphores can allow more than one thread to access a resource at once, though by default semaphores only allow one thread access at a time.

Basic semaphores

Semaphores have two methods, `down` and `up`. `down` decrements the resource count, while `up` increments it. `down` calls will block if the semaphore's current count would decrement below zero. This program gives a quick demonstration:

```
use Thread qw(yield);
use Thread::Semaphore;
my $semaphore = new Thread::Semaphore;
$GlobalVariable = 0;

$thr1 = new Thread \&sample_sub, 1;
$thr2 = new Thread \&sample_sub, 2;
$thr3 = new Thread \&sample_sub, 3;

sub sample_sub {
    my $SubNumber = shift @_;
    my $TryCount = 10;
    my $LocalCopy;
    sleep 1;
    while ($TryCount-- > 0) {
        $semaphore->down;
        $LocalCopy = $GlobalVariable;
        print "$TryCount tries left for sub $SubNumber (\$GlobalVariable is $GlobalVariable)";
        yield;
        sleep 2;
        $LocalCopy++;
        $GlobalVariable = $LocalCopy;
        $semaphore->up;
    }
}
```

The three invocations of the subroutine all operate in sync. The semaphore, though, makes sure that only one thread is accessing the global variable at once.

Advanced Semaphores

By default, semaphores behave like locks, letting only one thread `down()` them at a time. However, there are other uses for semaphores.

Each semaphore has a counter attached to it. `down()` decrements the counter and `up()` increments the counter. By default, semaphores are created with the counter set to one, `down()` decrements by one, and `up()` increments by one. If `down()` attempts to decrement the counter below zero, it blocks until the counter is large enough. Note that while a semaphore can be created with a starting count of zero, any `up()` or `down()` always changes the counter by at least one. `$semaphore->down(0)` is the

same as `$semaphore-down(1)`.

The question, of course, is why would you do something like this? Why create a semaphore with a starting count that's not one, or why decrement/increment it by more than one? The answer is resource availability. Many resources that you want to manage access for can be safely used by more than one thread at once.

For example, let's take a GUI driven program. It has a semaphore that it uses to synchronize access to the display, so only one thread is ever drawing at once. Handy, but of course you don't want any thread to start drawing until things are properly set up. In this case, you can create a semaphore with a counter set to zero, and up it when things are ready for drawing.

Semaphores with counters greater than one are also useful for establishing quotas. Say, for example, that you have a number of threads that can do I/O at once. You don't want all the threads reading or writing at once though, since that can potentially swamp your I/O channels, or deplete your process' quota of filehandles. You can use a semaphore initialized to the number of concurrent I/O requests (or open files) that you want at any one time, and have your threads quietly block and unblock themselves.

Larger increments or decrements are handy in those cases where a thread needs to check out or return a number of resources at once.

Attributes: Restricting Access To Subroutines

In addition to synchronizing access to data or resources, you might find it useful to synchronize access to subroutines. You may be accessing a singular machine resource (perhaps a vector processor), or find it easier to serialize calls to a particular subroutine than to have a set of locks and semaphores.

One of the additions to Perl 5.005 is subroutine attributes. The Thread package uses these to provide several flavors of serialization. It's important to remember that these attributes are used in the compilation phase of your program so you can't change a subroutine's behavior while your program is actually running.

Subroutine Locks

The basic subroutine lock looks like this:

```
sub test_sub :locked {
}
```

This ensures that only one thread will be executing this subroutine at any one time. Once a thread calls this subroutine, any other thread that calls it will block until the thread in the subroutine exits it. A more elaborate example looks like this:

```
use Thread qw(yield);

new Thread \&thread_sub, 1;
new Thread \&thread_sub, 2;
new Thread \&thread_sub, 3;
new Thread \&thread_sub, 4;

sub sync_sub :locked {
    my $CallingThread = shift @_;
    print "In sync_sub for thread $CallingThread\n";
    yield;
    sleep 3;
    print "Leaving sync_sub for thread $CallingThread\n";
}

sub thread_sub {
    my $ThreadID = shift @_;
    print "Thread $ThreadID calling sync_sub\n";
    sync_sub($ThreadID);
    print "$ThreadID is done with sync_sub\n";
}
```

```
}
```

The `locked` attribute tells perl to lock `sync_sub()`, and if you run this, you can see that only one thread is in it at any one time.

Methods

Locking an entire subroutine can sometimes be overkill, especially when dealing with Perl objects. When calling a method for an object, for example, you want to serialize calls to a method, so that only one thread will be in the subroutine for a particular object, but threads calling that subroutine for a different object aren't blocked. The `method` attribute indicates whether the subroutine is really a method.

```
use Thread;

sub tester {
    my $thrn timer = shift @_;
    my $bar = new Foo;
    foreach (1..10) {
        print "$thrn timer calling per_object\n";
        $bar->per_object($thrn timer);
        print "$thrn timer out of per_object\n";
        yield;
        print "$thrn timer calling one_at_a_time\n";
        $bar->one_at_a_time($thrn timer);
        print "$thrn timer out of one_at_a_time\n";
        yield;
    }
}

foreach my $thrn timer (1..10) {
    new Thread \&tester, $thrn timer;
}

package Foo;
sub new {
    my $class = shift @_;
    return bless [], $class;
}

sub per_object :locked :method {
    my ($class, $thrn timer) = @_;
    print "In per_object for thread $thrn timer\n";
    yield;
    sleep 2;
    print "Exiting per_object for thread $thrn timer\n";
}

sub one_at_a_time :locked {
    my ($class, $thrn timer) = @_;
    print "In one_at_a_time for thread $thrn timer\n";
    yield;
    sleep 2;
    print "Exiting one_at_a_time for thread $thrn timer\n";
}
```

As you can see from the output (omitted for brevity; it's 800 lines) all the threads can be in `per_object()` simultaneously, but only one thread is ever in `one_at_a_time()` at once.

Locking A Subroutine

You can lock a subroutine as you would lock a variable. Subroutine locks work the same as specifying a `locked` attribute for the subroutine, and block all access to the subroutine for other threads until the lock goes out of scope. When the subroutine isn't locked, any number of threads can be in it at once, and getting a lock on a subroutine doesn't affect threads already in the subroutine. Getting a lock on a subroutine looks like this:

```
lock(\&sub_to_lock);
```

Simple enough. Unlike the `locked` attribute, which is a compile time option, locking and unlocking a subroutine can be done at runtime at your discretion. There is some runtime penalty to using `lock(\&sub)` instead of the `locked` attribute, so make sure you're choosing the proper method to do the locking.

You'd choose `lock(\&sub)` when writing modules and code to run on both threaded and unthreaded Perl, especially for code that will run on 5.004 or earlier Perls. In that case, it's useful to have subroutines that should be serialized lock themselves if they're running threaded, like so:

```
package Foo;
use Config;
$Running_Threaded = 0;

BEGIN { $Running_Threaded = $Config{'usethreads'} }

sub sub1 { lock(\&sub1) if $Running_Threaded }
```

This way you can ensure single-threadedness regardless of which version of Perl you're running.

General Thread Utility Routines

We've covered the workhorse parts of Perl's threading package, and with these tools you should be well on your way to writing threaded code and packages. There are a few useful little pieces that didn't really fit in anyplace else.

What Thread Am I In?

The `Thread-self` method provides your program with a way to get an object representing the thread it's currently in. You can use this object in the same way as the ones returned from the thread creation.

Thread IDs

`tid()` is a thread object method that returns the thread ID of the thread the object represents. Thread IDs are integers, with the main thread in a program being 0. Currently Perl assigns a unique `tid` to every thread ever created in your program, assigning the first thread to be created a `tid` of 1, and increasing the `tid` by 1 for each new thread that's created.

Are These Threads The Same?

The `equal()` method takes two thread objects and returns true if the objects represent the same thread, and false if they don't.

What Threads Are Running?

`Thread-list` returns a list of thread objects, one for each thread that's currently running. Handy for a number of things, including cleaning up at the end of your program:

```
# Loop through all the threads
foreach $thr (Thread->list) {
    # Don't join the main thread or ourselves
    if ($thr->tid && !Thread::equal($thr, Thread->self)) {
        $thr->join;
    }
}
```

The example above is just for illustration. It isn't strictly necessary to join all the threads you create, since Perl detaches all the threads before it exits.

A Complete Example

Confused yet? It's time for an example program to show some of the things we've covered. This program finds prime numbers using threads.

```
1  #!/usr/bin/perl -w
2  # prime-pthread, courtesy of Tom Christiansen
3
4  use strict;
5
6  use Thread;
7  use Thread::Queue;
8
9  my $stream = new Thread::Queue;
10 my $kid     = new Thread(\&check_num, $stream, 2);
11
12 for my $i ( 3 .. 1000 ) {
13     $stream->enqueue($i);
14 }
15
16 $stream->enqueue(undef);
17 $kid->join();
18
19 sub check_num {
20     my ($upstream, $cur_prime) = @_;
21     my $kid;
22     my $downstream = new Thread::Queue;
23     while (my $num = $upstream->dequeue) {
24         next unless $num % $cur_prime;
25         if ($kid) {
26             $downstream->enqueue($num);
27         } else {
28             print "Found prime $num\n";
29             $kid = new Thread(\&check_num, $downstream, $num);
30         }
31     }
32     $downstream->enqueue(undef) if $kid;
33     $kid->join() if $kid;
34 }
```

This program uses the pipeline model to generate prime numbers. Each thread in the pipeline has an input queue that feeds numbers to be checked, a prime number that it's responsible for, and an output queue that it funnels numbers that have failed the check into. If the thread has a number that's failed its check and there's no child thread, then the thread must have found a new prime number. In that case, a new child thread is created for that prime and stuck on the end of the pipeline.

This probably sounds a bit more confusing than it really is, so let's go through this program piece by piece and see what it does. (For those of you who might be trying to remember exactly what a prime number is, it's a number that's only evenly divisible by itself and 1)

The bulk of the work is done by the `check_num()` subroutine, which takes a reference to its input queue and a prime number that it's responsible for. After pulling in the input queue and the prime that the subroutine's checking (line 20), we create a new queue (line 22) and reserve a scalar for the thread that we're likely to create later (line 21).

The while loop from lines 23 to line 31 grabs a scalar off the input queue and checks against the prime this thread is responsible for. Line 24 checks to see if there's a remainder when we modulo the number to be

checked against our prime. If there is one, the number must not be evenly divisible by our prime, so we need to either pass it on to the next thread if we've created one (line 26) or create a new thread if we haven't.

The new thread creation is line 29. We pass on to it a reference to the queue we've created, and the prime number we've found.

Finally, once the loop terminates (because we got a 0 or undef in the queue, which serves as a note to die), we pass on the notice to our child and wait for it to exit if we've created a child (Lines 32 and 37).

Meanwhile, back in the main thread, we create a queue (line 9) and the initial child thread (line 10), and pre-seed it with the first prime: 2. Then we queue all the numbers from 3 to 1000 for checking (lines 12–14), then queue a die notice (line 16) and wait for the first child thread to terminate (line 17). Because a child won't die until its child has died, we know that we're done once we return from the join.

That's how it works. It's pretty simple; as with many Perl programs, the explanation is much longer than the program.

Conclusion

A complete thread tutorial could fill a book (and has, many times), but this should get you well on your way. The final authority on how Perl's threads behave is the documentation bundled with the Perl distribution, but with what we've covered in this article, you should be well on your way to becoming a threaded Perl expert.

Bibliography

Here's a short bibliography courtesy of Jürgen Christoffel:

Introductory Texts

Birrell, Andrew D. An Introduction to Programming with Threads. Digital Equipment Corporation, 1989, DEC–SRC Research Report #35 online as <http://www.research.digital.com/SRC/staff/birrell/bib.html> (highly recommended)

Robbins, Kay. A., and Steven Robbins. Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading. Prentice–Hall, 1996.

Lewis, Bill, and Daniel J. Berg. Multithreaded Programming with Pthreads. Prentice Hall, 1997, ISBN 0–13–443698–9 (a well–written introduction to threads).

Nelson, Greg (editor). Systems Programming with Modula–3. Prentice Hall, 1991, ISBN 0–13–590464–1.

Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. Pthreads Programming. O'Reilly & Associates, 1996, ISBN 156592–115–1 (covers POSIX threads).

OS–Related References

Boykin, Joseph, David Kirschen, Alan Langerman, and Susan LoVerso. Programming under Mach. Addison–Wesley, 1994, ISBN 0–201–52739–1.

Tanenbaum, Andrew S. Distributed Operating Systems. Prentice Hall, 1995, ISBN 0–13–219908–4 (great textbook).

Silberschatz, Abraham, and Peter B. Galvin. Operating System Concepts, 4th ed. Addison–Wesley, 1995, ISBN 0–201–59292–4

Other References

Arnold, Ken and James Gosling. The Java Programming Language, 2nd ed. Addison–Wesley, 1998, ISBN 0–201–31006–6.

Le Sergent, T. and B. Berthomieu. "Incremental MultiThreaded Garbage Collection on Virtually Shared Memory Architectures" in Memory Management: Proc. of the International Workshop IWMM 92, St. Malo, France, September 1992, Yves Bekkers and Jacques Cohen, eds. Springer, 1992, ISBN 3540–55940–X (real–life thread applications).

Acknowledgements

Thanks (in no particular order) to Chaim Frenkel, Steve Fink, Gurusamy Sarathy, Ilya Zakharevich, Benjamin Sugars, Jürgen Christoffel, Joshua Pritikin, and Alan Burlison, for their help in reality-checking and polishing this article. Big thanks to Tom Christiansen for his rewrite of the prime number generator.

AUTHOR

Dan Sugalski <sugalskd@ous.edu>

Copyrights

This article originally appeared in The Perl Journal #10, and is copyright 1998 The Perl Journal. It appears courtesy of Jon Orwant and The Perl Journal. This document may be distributed under the same terms as Perl itself.

NAME

perlpacktut – tutorial on pack and unpack

DESCRIPTION

`pack` and `unpack` are two functions for transforming data according to a user-defined template, between the guarded way Perl stores values and some well-defined representation as might be required in the environment of a Perl program. Unfortunately, they're also two of the most misunderstood and most often overlooked functions that Perl provides. This tutorial will demystify them for you.

The Basic Principle

Most programming languages don't shelter the memory where variables are stored. In C, for instance, you can take the address of some variable, and the `sizeof` operator tells you how many bytes are allocated to the variable. Using the address and the size, you may access the storage to your heart's content.

In Perl, you just can't access memory at random, but the structural and representational conversion provided by `pack` and `unpack` is an excellent alternative. The `pack` function converts values to a byte sequence containing representations according to a given specification, the so-called "template" argument. `unpack` is the reverse process, deriving some values from the contents of a string of bytes. (Be cautioned, however, that not all that has been packed together can be neatly unpacked – a very common experience as seasoned travellers are likely to confirm.)

Why, you may ask, would you need a chunk of memory containing some values in binary representation? One good reason is input and output accessing some file, a device, or a network connection, whereby this binary representation is either forced on you or will give you some benefit in processing. Another cause is passing data to some system call that is not available as a Perl function: `syscall` requires you to provide parameters stored in the way it happens in a C program. Even text processing (as shown in the next section) may be simplified with judicious usage of these two functions.

To see how (un)packing works, we'll start with a simple template code where the conversion is in low gear: between the contents of a byte sequence and a string of hexadecimal digits. Let's use `unpack`, since this is likely to remind you of a dump program, or some desperate last message unfortunate programs are wont to throw at you before they expire into the wild blue yonder. Assuming that the variable `$mem` holds a sequence of bytes that we'd like to inspect without assuming anything about its meaning, we can write

```
my( $hex ) = unpack( 'H*', $mem );
print "$hex\n";
```

whereupon we might see something like this, with each pair of hex digits corresponding to a byte:

```
41204d4114e204120504c414e204120434114e414c2050414e414d41
```

What was in this chunk of memory? Numbers, characters, or a mixture of both? Assuming that we're on a computer where ASCII (or some similar) encoding is used: hexadecimal values in the range 0x40 – 0x5A indicate an uppercase letter, and 0x20 encodes a space. So we might assume it is a piece of text, which some are able to read like a tabloid; but others will have to get hold of an ASCII table and relive that firstgrader feeling. Not caring too much about which way to read this, we note that `unpack` with the template code `H` converts the contents of a sequence of bytes into the customary hexadecimal notation. Since "a sequence of" is a pretty vague indication of quantity, `H` has been defined to convert just a single hexadecimal digit unless it is followed by a repeat count. An asterisk for the repeat count means to use whatever remains.

The inverse operation – packing byte contents from a string of hexadecimal digits – is just as easily written. For instance:

```
my $s = pack( 'H2' x 10, map { "3$_" } ( 0..9 ) );
print "$s\n";
```

Since we feed a list of ten 2-digit hexadecimal strings to `pack`, the `pack` template should contain ten `pack` codes. If this is run on a computer with ASCII character coding, it will print 0123456789.

Packing Text

Let's suppose you've got to read in a data file like this:

Date	Description	Income	Expenditure
01/24/2001	Ahmed's Camel Emporium		1147.99
01/28/2001	Flea spray		24.99
01/29/2001	Camel rides to tourists	235.00	

How do we do it? You might think first to use `split`; however, since `split` collapses blank fields, you'll never know whether a record was income or expenditure. Oops. Well, you could always use `substr`:

```
while (<>) {
    my $date   = substr($_, 0, 11);
    my $desc   = substr($_, 12, 27);
    my $income = substr($_, 40, 7);
    my $expend = substr($_, 52, 7);
    ...
}
```

It's not really a barrel of laughs, is it? In fact, it's worse than it may seem; the eagle-eyed may notice that the first field should only be 10 characters wide, and the error has propagated right through the other numbers – which we've had to count by hand. So it's error-prone as well as horribly unfriendly.

Or maybe we could use regular expressions:

```
while (<>) {
    my($date, $desc, $income, $expend) =
        m|(\d\d/\d\d/\d\d\d{4}) (.{27}) (.{7}) (.*)|;
    ...
}
```

Urgh. Well, it's a bit better, but – well, would you want to maintain that?

Hey, isn't Perl supposed to make this sort of thing easy? Well, it does, if you use the right tools. `pack` and `unpack` are designed to help you out when dealing with fixed-width data like the above. Let's have a look at a solution with `unpack`:

```
while (<>) {
    my($date, $desc, $income, $expend) = unpack("A10xA27xA7A*", $_);
    ...
}
```

That looks a bit nicer; but we've got to take apart that weird template. Where did I pull that out of?

OK, let's have a look at some of our data again; in fact, we'll include the headers, and a handy ruler so we can keep track of where we are.

1	2	3	4	5
1234567890123456789012345678901234567890123456789012345678				
Date	Description		Income	Expenditure
01/28/2001	Flea spray			24.99
01/29/2001	Camel rides to tourists		235.00	

From this, we can see that the date column stretches from column 1 to column 10 – ten characters wide. The `pack`-ese for "character" is `A`, and ten of them are `A10`. So if we just wanted to extract the dates, we could say this:

```
my($date) = unpack("A10", $_);
```

OK, what's next? Between the date and the description is a blank column; we want to skip over that. The `x` template means "skip forward", so we want one of those. Next, we have another batch of characters, from 12

to 38. That's 27 more characters, hence A27. (Don't make the fencepost error – there are 27 characters between 12 and 38, not 26. Count 'em!)

Now we skip another character and pick up the next 7 characters:

```
my($date,$description,$income) = unpack("A10xA27xA7", $_);
```

Now comes the clever bit. Lines in our ledger which are just income and not expenditure might end at column 46. Hence, we don't want to tell our unpack pattern that we **need** to find another 12 characters; we'll just say "if there's anything left, take it". As you might guess from regular expressions, that's what the `*` means: "use everything remaining".

- Be warned, though, that unlike regular expressions, if the unpack template doesn't match the incoming data, Perl will scream and die.

Hence, putting it all together:

```
my($date,$description,$income,$expend) = unpack("A10xA27xA7A*", $_);
```

Now, that's our data parsed. I suppose what we might want to do now is total up our income and expenditure, and add another line to the end of our ledger – in the same format – saying how much we've brought in and how much we've spent:

```
while (<>) {
    my($date, $desc, $income, $expend) = unpack("A10xA27xA7xA*", $_);
    $tot_income += $income;
    $tot_expend += $expend;
}

$tot_income = sprintf("%.2f", $tot_income); # Get them into
$tot_expend = sprintf("%.2f", $tot_expend); # "financial" format

$date = POSIX::strftime("%m/%d/%Y", localtime);

# OK, let's go:

print pack("A10xA27xA7xA*", $date, "Totals", $tot_income, $tot_expend);
```

Oh, hmm. That didn't quite work. Let's see what happened:

```
01/24/2001 Ahmed's Camel Emporium          1147.99
01/28/2001 Flea spray                        24.99
01/29/2001 Camel rides to tourists          1235.00
03/23/2001Totals                            1235.001172.98
```

OK, it's a start, but what happened to the spaces? We put `x`, didn't we? Shouldn't it skip forward? Let's look at what [pack](#) says:

```
x    A null byte.
```

Urgh. No wonder. There's a big difference between "a null byte", character zero, and "a space", character 32. Perl's put something between the date and the description – but unfortunately, we can't see it!

What we actually need to do is expand the width of the fields. The `A` format pads any non-existent characters with spaces, so we can use the additional spaces to line up our fields, like this:

```
print pack("A11 A28 A8 A*", $date, "Totals", $tot_income, $tot_expend);
```

(Note that you can put spaces in the template to make it more readable, but they don't translate to spaces in the output.) Here's what we got this time:

```
01/24/2001 Ahmed's Camel Emporium          1147.99
01/28/2001 Flea spray                        24.99
01/29/2001 Camel rides to tourists          1235.00
03/23/2001 Totals                            1235.00 1172.98
```

That's a bit better, but we still have that last column which needs to be moved further over. There's an easy way to fix this up: unfortunately, we can't get `pack` to right-justify our fields, but we can get `sprintf` to do it:

```
$tot_income = sprintf("%.2f", $tot_income);
$tot_expend = sprintf("%12.2f", $tot_expend);
$date = POSIX::strftime("%m/%d/%Y", localtime);
print pack("A11 A28 A8 A*", $date, "Totals", $tot_income, $tot_expend);
```

This time we get the right answer:

```
01/28/2001 Flea spray                      24.99
01/29/2001 Camel rides to tourists        1235.00
03/23/2001 Totals                        1235.00    1172.98
```

So that's how we consume and produce fixed-width data. Let's recap what we've seen of `pack` and `unpack` so far:

- Use `pack` to go from several pieces of data to one fixed-width version; use `unpack` to turn a fixed-width-format string into several pieces of data.
- The `pack` format `A` means "any character"; if you're packing and you've run out of things to pack, `pack` will fill the rest up with spaces.
- `x` means "skip a byte" when unpacking; when packing, it means "introduce a null byte" – that's probably not what you mean if you're dealing with plain text.
- You can follow the formats with numbers to say how many characters should be affected by that format: `A12` means "take 12 characters"; `x6` means "skip 6 bytes" or "character 0, 6 times".
- Instead of a number, you can use `*` to mean "consume everything else left".

Warning: when packing multiple pieces of data, `*` only means "consume all of the current piece of data". That's to say

```
pack("A*A*", $one, $two)
```

packs all of `$one` into the first `A*` and then all of `$two` into the second. This is a general principle: each format character corresponds to one piece of data to be packed.

Packing Numbers

So much for textual data. Let's get onto the meaty stuff that `pack` and `unpack` are best at: handling binary formats for numbers. There is, of course, not just one binary format – life would be too simple – but Perl will do all the finicky labor for you.

Integers

Packing and unpacking numbers implies conversion to and from some *specific* binary representation. Leaving floating point numbers aside for the moment, the salient properties of any such representation are:

- the number of bytes used for storing the integer,
- whether the contents are interpreted as a signed or unsigned number,
- the byte ordering: whether the first byte is the least or most significant byte (or: little-endian or big-endian, respectively).

So, for instance, to pack 20302 to a signed 16 bit integer in your computer's representation you write

```
my $ps = pack( 's', 20302 );
```

Again, the result is a string, now containing 2 bytes. If you print this string (which is, generally, not recommended) you might see `ON` or `NO` (depending on your system's byte ordering) – or something entirely different if your computer doesn't use ASCII character encoding. Unpacking `$ps` with the same template returns the original integer value:

It would be nice if we could do this in one fell swoop: unpack a short, back up a little, and then unpack 2 bytes. Since Perl *is* nice, it proffers the template code X to back up one byte. Putting this all together, we may now write:

```
my( $ip, $cs,
    $flags,$fl,$fh,
    $ax,$al,$ah, $bx,$bl,$bh, $cx,$cl,$ch, $dx,$dl,$dh,
    $si, $di, $bp, $ds, $es ) =
    unpack( 'v2' . ('vXXCC' x 5) . 'v5', $frame );
```

We've taken some pains to construct the template so that it matches the contents of our frame buffer. Otherwise we'd either get undefined values, or unpack could not unpack all. If pack runs out of items, it will supply null strings (which are coerced into zeroes whenever the pack code says so).

How to Eat an Egg on a Net

The pack code for big-endian (high order byte at the lowest address) is n for 16 bit and N for 32 bit integers. You use these codes if you know that your data comes from a compliant architecture, but, surprisingly enough, you should also use these pack codes if you exchange binary data, across the network, with some system that you know next to nothing about. The simple reason is that this order has been chosen as the *network order*, and all standard-fearing programs ought to follow this convention. (This is, of course, a stern backing for one of the Lilliputian parties and may well influence the political development there.) So, if the protocol expects you to send a message by sending the length first, followed by just so many bytes, you could write:

```
my $buf = pack( 'N', length( $msg ) ) . $msg;
```

or even:

```
my $buf = pack( 'NA*', length( $msg ), $msg );
```

and pass \$buf to your send routine. Some protocols demand that the count should include the length of the count itself: then just add 4 to the data length. (But make sure to read "[Lengths and Widths](#)" before you really code this!)

Floating point Numbers

For packing floating point numbers you have the choice between the pack codes f and d which pack into (or unpack from) single-precision or double-precision representation as it is provided by your system. (There is no such thing as a network representation for reals, so if you want to send your real numbers across computer boundaries, you'd better stick to ASCII representation, unless you're absolutely sure what's on the other end of the line.)

Exotic Templates

Bit Strings

Bits are the atoms in the memory world. Access to individual bits may have to be used either as a last resort or because it is the most convenient way to handle your data. Bit string (un)packing converts between strings containing a series of 0 and 1 characters and a sequence of bytes each containing a group of 8 bits. This is almost as simple as it sounds, except that there are two ways the contents of a byte may be written as a bit string. Let's have a look at an annotated byte:

```

  7 6 5 4 3 2 1 0
+-----+
| 1 0 0 0 1 1 0 0 |
+-----+
  MSB                LSB
```

It's egg-eating all over again: Some think that as a bit string this should be written "10001100" i.e. beginning with the most significant bit, others insist on "00110001". Well, Perl isn't biased, so that's why we have two bit string codes:

```
$byte = pack( 'B8', '10001100' ); # start with MSB
$byte = pack( 'b8', '00110001' ); # start with LSB
```

It is not possible to pack or unpack bit fields – just integral bytes. `pack` always starts at the next byte boundary and "rounds up" to the next multiple of 8 by adding zero bits as required. (If you do want bit fields, there is [vec](#). Or you could implement bit field handling at the character string level, using `split`, `substr`, and concatenation on unpacked bit strings.)

To illustrate unpacking for bit strings, we'll decompose a simple status register (a "-" stands for a "reserved" bit):

```
+-----+-----+
| S Z - A - P - C | - - - - O D I T |
+-----+-----+
MSB                LSB MSB                LSB
```

Converting these two bytes to a string can be done with the `unpack` template `'b16'`. To obtain the individual bit values from the bit string we use `split` with the "empty" separator pattern which dissects into individual characters. Bit values from the "reserved" positions are simply assigned to `undef`, a convenient notation for "I don't care where this goes".

```
($carry, undef, $parity, undef, $auxcarry, undef, $sign,
 $trace, $interrupt, $direction, $overflow) =
    split( //, unpack( 'b16', $status ) );
```

We could have used an `unpack` template `'b12'` just as well, since the last 4 bits can be ignored anyway.

Uuencoding

Another odd-man-out in the template alphabet is `u`, which packs an "uuencoded string". ("uu" is short for Unix-to-Unix.) Chances are that you won't ever need this encoding technique which was invented to overcome the shortcomings of old-fashioned transmission mediums that do not support other than simple ASCII data. The essential recipe is simple: Take three bytes, or 24 bits. Split them into 4 six-packs, adding a space (0x20) to each. Repeat until all of the data is blended. Fold groups of 4 bytes into lines no longer than 60 and garnish them in front with the original byte count (incremented by 0x20) and a "\n" at the end. – The pack chef will prepare this for you, a la minute, when you select pack code `u` on the menu:

```
my $uubuf = pack( 'u', $bindat );
```

A repeat count after `u` sets the number of bytes to put into an uuencoded line, which is the maximum of 45 by default, but could be set to some (smaller) integer multiple of three. `unpack` simply ignores the repeat count.

Doing Sums

An even stranger template code is `%<number>`. First, because it's used as a prefix to some other template code. Second, because it cannot be used in `pack` at all, and third, in `unpack`, doesn't return the data as defined by the template code it precedes. Instead it'll give you an integer of *number* bits that is computed from the data value by doing sums. For numeric `unpack` codes, no big feat is achieved:

```
my $buf = pack( 'iii', 100, 20, 3 );
print unpack( '%32i3', $buf ), "\n"; # prints 123
```

For string values, `%` returns the sum of the byte values saving you the trouble of a sum loop with `substr` and `ord`:

```
print unpack( '%32A*', "\x01\x10" ), "\n"; # prints 17
```

Although the `%` code is documented as returning a "checksum": don't put your trust in such values! Even when applied to a small number of bytes, they won't guarantee a noticeable Hamming distance.

In connection with `b` or `B`, `%` simply adds bits, and this can be put to good use to count set bits efficiently:

```
my $bitcount = unpack( '%32b*', $mask );
```

And an even parity bit can be determined like this:

```
my $evenparity = unpack( '%1b*', $mask );
```

Unicode

Unicode is a character set that can represent most characters in most of the world's languages, providing room for over one million different characters. Unicode 3.1 specifies 94,140 characters: The Basic Latin characters are assigned to the numbers 0 – 127. The Latin-1 Supplement with characters that are used in several European languages is in the next range, up to 255. After some more Latin extensions we find the character sets from languages using non-Roman alphabets, interspersed with a variety of symbol sets such as currency symbols, Zapf Dingbats or Braille. (You might want to visit www.unicode.org for a look at some of them – my personal favourites are Telugu and Kannada.)

The Unicode character sets associates characters with integers. Encoding these numbers in an equal number of bytes would more than double the requirements for storing texts written in Latin alphabets. The UTF-8 encoding avoids this by storing the most common (from a western point of view) characters in a single byte while encoding the rarer ones in three or more bytes.

So what has this got to do with pack? Well, if you want to convert between a Unicode number and its UTF-8 representation you can do so by using template code U. As an example, let's produce the UTF-8 representation of the Euro currency symbol (code number 0x20AC):

```
$UTF8{Euro} = pack( 'U', 0x20AC );
```

Inspecting `$UTF8{Euro}` shows that it contains 3 bytes: `"\xe2\x82\xac"`. The round trip can be completed with unpack:

```
$Unicode{Euro} = unpack( 'U', $UTF8{Euro} );
```

Usually you'll want to pack or unpack UTF-8 strings:

```
# pack and unpack the Hebrew alphabet
my $alefbet = pack( 'U*', 0x05d0..0x05ea );
my @hebrew = unpack( 'U*', $utf );
```

Another Portable Binary Encoding

The pack code w has been added to support a portable binary data encoding scheme that goes way beyond simple integers. (Details can be found at Casbah.org, the Scarab project.) A BER (Binary Encoded Representation) compressed unsigned integer stores base 128 digits, most significant digit first, with as few digits as possible. Bit eight (the high bit) is set on each byte except the last. There is no size limit to BER encoding, but Perl won't go to extremes.

```
my $berbuf = pack( 'w*', 1, 128, 128+1, 128*128+127 );
```

A hex dump of `$berbuf`, with spaces inserted at the right places, shows 01 8100 8101 81807F. Since the last byte is always less than 128, unpack knows where to stop.

Lengths and Widths

String Lengths

In the previous section we've seen a network message that was constructed by prefixing the binary message length to the actual message. You'll find that packing a length followed by so many bytes of data is a frequently used recipe since appending a null byte won't work if a null byte may be part of the data. Here is an example where both techniques are used: after two null terminated strings with source and destination address, a Short Message (to a mobile phone) is sent after a length byte:

```
my $msg = pack( 'Z*Z*CA*', $src, $dst, length( $sm ), $sm );
```

Unpacking this message can be done with the same template:

```
( $src, $dst, $len, $sm ) = unpack( 'Z*Z*CA*', $msg );
```

There's a subtle trap lurking in the offing: Adding another field after the Short Message (in variable `$sm`) is all right when packing, but this cannot be unpacked naively:

```
# pack a message
my $msg = pack( 'Z*Z*CA*C', $src, $dst, length( $sm ), $sm, $prio );

# unpack fails - $prio remains undefined!
( $src, $dst, $len, $sm, $prio ) = unpack( 'Z*Z*CA*C', $msg );
```

The pack code `A*` gobbles up all remaining bytes, and `$prio` remains undefined! Before we let disappointment dampen the morale: Perl's got the trump card to make this trick too, just a little further up the sleeve. Watch this:

```
# pack a message: ASCIIIZ, ASCIIIZ, length/string, byte
my $msg = pack( 'Z* Z* C/A* C', $src, $dst, $sm, $prio );

# unpack
( $src, $dst, $sm, $prio ) = unpack( 'Z* Z* C/A* C', $msg );
```

Combining two pack codes with a slash (/) associates them with a single value from the argument list. In pack, the length of the argument is taken and packed according to the first code while the argument itself is added after being converted with the template code after the slash. This saves us the trouble of inserting the length call, but it is in unpack where we really score: The value of the length byte marks the end of the string to be taken from the buffer. Since this combination doesn't make sense except when the second pack code isn't `a*`, `A*` or `Z*`, Perl won't let you.

The pack code preceding / may be anything that's fit to represent a number: All the numeric binary pack codes, and even text codes such as `A4` or `Z*`:

```
# pack/unpack a string preceded by its length in ASCII
my $buf = pack( 'A4/A*', "Humpty-Dumpty" );
# unpack $buf: '13 Humpty-Dumpty'
my $txt = unpack( 'A4/A*', $buf );
```

/ is not implemented in Perls before 5.6, so if your code is required to work on older Perls you'll need to unpack('Z* Z* C') to get the length, then use it to make a new unpack string. For example

```
# pack a message: ASCIIIZ, ASCIIIZ, length, string, byte (5.005 compatible)
my $msg = pack( 'Z* Z* C A* C', $src, $dst, length $sm, $sm, $prio );

# unpack
( undef, undef, $len) = unpack( 'Z* Z* C', $msg );
($src, $dst, $sm, $prio) = unpack ( "Z* Z* x A$len C", $msg );
```

But that second unpack is rushing ahead. It isn't using a simple literal string for the template. So maybe we should introduce...

Dynamic Templates

So far, we've seen literals used as templates. If the list of pack items doesn't have fixed length, an expression constructing the template has to be used. Here's an example: To store named string values in a way that can be conveniently parsed by a C program, we create a sequence of names and null terminated ASCII strings, with = between the name and the value, followed by an additional delimiting null byte. Here's how:

```
my $env = pack( 'A*A*Z*' x keys( %Env ) . 'C',
               map( { ( $_, '=', $Env{$_} ) } keys( %Env ) ), 0 );
```

Let's examine the cogs of this byte mill, one by one. There's the map call, creating the items we intend to stuff into the `$env` buffer: to each key (in `$_`) it adds the = separator and the hash entry value. Each triplet is packed with the template code sequence `A*A*Z*` that is multiplied with the number of keys. (Yes, that's what the keys function returns in scalar context.) To get the very last null byte, we add a `@` at the end of the pack list, to be packed with `C`. (Attentive readers may have noticed that we could have omitted the 0.)

For the reverse operation, we'll have to determine the number of items in the buffer before we can let unpack rip it apart:

```
my $n = $env =~ tr/\0// - 1;
my %env = map( split( /=/, $_ ), unpack( 'Z*' x $n, $env ) );
```

The `tr` counts the null bytes. The `unpack` call returns a list of name–value pairs each of which is taken apart in the `map` block.

Packing and Unpacking C Structures

In previous sections we have seen how to pack numbers and character strings. If it were not for a couple of snags we could conclude this section right away with the terse remark that C structures don't contain anything else, and therefore you already know all there is to it. Sorry, no: read on, please.

The Alignment Pit

In the consideration of speed against memory requirements the balance has been tilted in favor of faster execution. This has influenced the way C compilers allocate memory for structures: On architectures where a 16-bit or 32-bit operand can be moved faster between places in memory, or to or from a CPU register, if it is aligned at an even or multiple-of-four or even at a multiple-of-eight address, a C compiler will give you this speed benefit by stuffing extra bytes into structures. If you don't cross the C shoreline this is not likely to cause you any grief (although you should care when you design large data structures, or you want your code to be portable between architectures (you do want that, don't you?)).

To see how this affects `pack` and `unpack`, we'll compare these two C structures:

```
typedef struct {
    char    c1;
    short   s;
    char    c2;
    long    l;
} gappy_t;

typedef struct {
    long    l;
    short   s;
    char    c1;
    char    c2;
} dense_t;
```

Typically, a C compiler allocates 12 bytes to a `gappy_t` variable, but requires only 8 bytes for a `dense_t`. After investigating this further, we can draw memory maps, showing where the extra 4 bytes are hidden:

```

0          +4          +8          +12
+---+---+---+---+---+---+---+---+
|c1|xx|  s  |c2|xx|xx|xx|      1      |      xx = fill byte
+---+---+---+---+---+---+---+---+
gappy_t

0          +4          +8
+---+---+---+---+---+---+
|      l      |  h  |c1|c2|
+---+---+---+---+---+---+
dense_t
```

And that's where the first quirk strikes: `pack` and `unpack` templates have to be stuffed with `x` codes to get those extra fill bytes.

The natural question: "Why can't Perl compensate for the gaps?" warrants an answer. One good reason is that C compilers might provide (non-ANSI) extensions permitting all sorts of fancy control over the way structures are aligned, even at the level of an individual structure field. And, if this were not enough, there is

an insidious thing called union where the amount of fill bytes cannot be derived from the alignment of the next item alone.

OK, so let's bite the bullet. Here's one way to get the alignment right by inserting template codes x, which don't take a corresponding item from the list:

```
my $gappy = pack( 'cxs cxxx l!', $c1, $s, $c2, $l );
```

Note the ! after l: We want to make sure that we pack a long integer as it is compiled by our C compiler. And even now, it will only work for the platforms where the compiler aligns things as above. And somebody somewhere has a platform where it doesn't. [Probably a Cray, where shorts, ints and longs are all 8 bytes. :-)]

Counting bytes and watching alignments in lengthy structures is bound to be a drag. Isn't there a way we can create the template with a simple program? Here's a C program that does the trick:

```
#include <stdio.h>
#include <stddef.h>

typedef struct {
    char    fc1;
    short   fs;
    char    fc2;
    long    fl;
} gappy_t;

#define Pt(struct,field,tchar) \
    printf( "%d%s ", offsetof(struct,field), # tchar );

int main(){
    Pt( gappy_t, fc1, c );
    Pt( gappy_t, fs,  s! );
    Pt( gappy_t, fc2, c );
    Pt( gappy_t, fl,  l! );
    printf( "\n" );
}
```

The output line can be used as a template in a pack or unpack call:

```
my $gappy = pack( '@0c @2s! @4c @8l!', $c1, $s, $c2, $l );
```

Gee, yet another template code – as if we hadn't plenty. But @ saves our day by enabling us to specify the offset from the beginning of the pack buffer to the next item: This is just the value the `offsetof` macro (defined in `<stddef.h>`) returns when given a `struct` type and one of its field names ("member-designator" in C standardese).

Alignment, Take 2

I'm afraid that we're not quite through with the alignment catch yet. The hydra raises another ugly head when you pack arrays of structures:

```
typedef struct {
    short    count;
    char     glyph;
} cell_t;

typedef cell_t buffer_t[BUFLLEN];
```

Where's the catch? Padding is neither required before the first field `count`, nor between this and the next field `glyph`, so why can't we simply pack like this:

```
# something goes wrong here:
pack( 's!a' x @buffer,
```

```
map{ ( $_->{count}, $_->{glyph} ) } @buffer );
```

This packs 3*@buffer bytes, but it turns out that the size of `buffer_t` is four times BUFLen! The moral of the story is that the required alignment of a structure or array is propagated to the next higher level where we have to consider padding *at the end* of each component as well. Thus the correct template is:

```
pack( 's!ax' x @buffer,
      map{ ( $_->{count}, $_->{glyph} ) } @buffer );
```

Alignment, Take 3

And even if you take all the above into account, ANSI still lets this:

```
typedef struct {
    char    foo[2];
} foo_t;
```

vary in size. The alignment constraint of the structure can be greater than any of its elements. [And if you think that this doesn't affect anything common, dismember the next cellphone that you see. Many have ARM cores, and the ARM structure rules make `sizeof (foo_t) == 4`]

Pointers for How to Use Them

The title of this section indicates the second problem you may run into sooner or later when you pack C structures. If the function you intend to call expects a, say, `void * value`, you *cannot* simply take a reference to a Perl variable. (Although that value certainly is a memory address, it's not the address where the variable's contents are stored.)

Template code P promises to pack a "pointer to a fixed length string". Isn't this what we want? Let's try:

```
# allocate some storage and pack a pointer to it
my $memory = "\x00" x $size;
my $memptr = pack( 'P', $memory );
```

But wait: doesn't pack just return a sequence of bytes? How can we pass this string of bytes to some C code expecting a pointer which is, after all, nothing but a number? The answer is simple: We have to obtain the numeric address from the bytes returned by pack.

```
my $ptr = unpack( 'L!', $memptr );
```

Obviously this assumes that it is possible to typecast a pointer to an unsigned long and vice versa, which frequently works but should not be taken as a universal law. – Now that we have this pointer the next question is: How can we put it to good use? We need a call to some C function where a pointer is expected. The `read(2)` system call comes to mind:

```
ssize_t read(int fd, void *buf, size_t count);
```

After reading [perlfunc](#) explaining how to use `syscall` we can write this Perl function copying a file to standard output:

```
require 'syscall.ph';
sub cat($) {
    my $path = shift();
    my $size = -s $path;
    my $memory = "\x00" x $size; # allocate some memory
    my $ptr = unpack( 'L', pack( 'P', $memory ) );
    open( F, $path ) || die( "$path: cannot open ($!)\n" );
    my $fd = fileno(F);
    my $res = syscall( &SYS_read, fileno(F), $ptr, $size );
    print $memory;
    close( F );
}
```

This is neither a specimen of simplicity nor a paragon of portability but it illustrates the point: We are able to sneak behind the scenes and access Perl's otherwise well-guarded memory! (Important note: Perl's `syscall` does *not* require you to construct pointers in this roundabout way. You simply pass a string variable, and Perl forwards the address.)

How does `unpack` with `P` work? Imagine some pointer in the buffer about to be unpacked: If it isn't the null pointer (which will smartly produce the `undef` value) we have a start address – but then what? Perl has no way of knowing how long this "fixed length string" is, so it's up to you to specify the actual size as an explicit length after `P`.

```
my $mem = "abcdefghijklmn";
print unpack( 'P5', pack( 'P', $mem ) ); # prints "abcde"
```

As a consequence, `pack` ignores any number or `*` after `P`.

Now that we have seen `P` at work, we might as well give `p` a whirl. Why do we need a second template code for packing pointers at all? The answer lies behind the simple fact that an `unpack` with `p` promises a null-terminated string starting at the address taken from the buffer, and that implies a length for the data item to be returned:

```
my $buf = pack( 'p', "abc\x00efhijklmn" );
print unpack( 'p', $buf ); # prints "abc"
```

Albeit this is apt to be confusing: As a consequence of the length being implied by the string's length, a number after `pack` code `p` is a repeat count, not a length as after `P`.

Using `pack(. . . , $x)` with `P` or `p` to get the address where `$x` is actually stored must be used with circumspection. Perl's internal machinery considers the relation between a variable and that address as its very own private matter and doesn't really care that we have obtained a copy. Therefore:

- Do not use `pack` with `p` or `P` to obtain the address of variable that's bound to go out of scope (and thereby freeing its memory) before you are done with using the memory at that address.
- Be very careful with Perl operations that change the value of the variable. Appending something to the variable, for instance, might require reallocation of its storage, leaving you with a pointer into no-man's land.
- Don't think that you can get the address of a Perl variable when it is stored as an integer or double number! `pack('P', $x)` will force the variable's internal representation to string, just as if you had written something like `$x .= ''`.

It's safe, however, to `P`- or `p`-pack a string literal, because Perl simply allocates an anonymous variable.

Pack Recipes

Here are a collection of (possibly) useful canned recipes for `pack` and `unpack`:

```
# Convert IP address for socket functions
pack( "C4", split /\./, "123.4.5.6" );

# Count the bits in a chunk of memory (e.g. a select vector)
unpack( '%32b*', $mask );

# Determine the endianness of your system
$is_little_endian = unpack( 'c', pack( 's', 1 ) );
$is_big_endian = unpack( 'xc', pack( 's', 1 ) );

# Determine the number of bits in a native integer
$bits = unpack( '%32I!', ~0 );

# Prepare argument for the nanosleep system call
my $timespec = pack( 'L!L!', $secs, $nanosecs );
```

For a simple memory dump we unpack some bytes into just as many pairs of hex digits, and use `map` to

handle the traditional spacing – 16 bytes to a line:

```
my $i;
print map { ++$i % 16 ? "$_ " : "$_\n" }
    unpack( 'H2' x length( $mem ), $mem ),
    length( $mem ) % 16 ? "\n" : '';
```

Funnies Section

```
# Pulling digits out of nowhere...
print unpack( 'C', pack( 'x' ) ),
    unpack( '%B*', pack( 'A' ) ),
    unpack( 'H', pack( 'A' ) ),
    unpack( 'A', unpack( 'C', pack( 'A' ) ) ), "\n";

# One for the road ;- )
my $advice = pack( 'all u can in a van' );
```

Authors

Simon Cozens and Wolfgang Laun.

NAME

perlpod – the Plain Old Documentation format

DESCRIPTION

Pod is a simple-to-use markup language used for writing documentation for Perl, Perl programs, and Perl modules.

Translators are available for converting Pod to various formats like plain text, HTML, man pages, and more.

Pod markup consists of three basic kinds of paragraphs: *Ordinary Paragraph in ordinary/*, *Verbatim Paragraph in verbatim/*, and *Command Paragraph in command/*.

Ordinary Paragraph

Most paragraphs in your documentation will be ordinary blocks of text, like this one. You can simply type in your text without any markup whatsoever, and with just a blank line before and after. When it gets formatted, it will undergo minimal formatting, like being rewrapped, probably put into a proportionally spaced font, and maybe even justified.

You can use formatting codes in ordinary paragraphs, for **bold**, *italic*, code-style, [hyperlinks/perlfaq](#), and more. Such codes are explained in the "[Formatting Codes in Formatting Codes](#)" section, below.

Verbatim Paragraph

Verbatim paragraphs are usually used for presenting a codeblock or other text which does not require any special parsing or formatting, and which shouldn't be wrapped.

A verbatim paragraph is distinguished by having its first character be a space or a tab. (And commonly, all its lines begin with spaces and/or tabs.) It should be reproduced exactly, with tabs assumed to be on 8-column boundaries. There are no special formatting codes, so you can't italicize or anything like that. A \ means \, and nothing else.

Command Paragraph

A command paragraph is used for special treatment of whole chunks of text, usually as headings or parts of lists.

All command paragraphs (which are typically only one line long) start with "=", followed by an identifier, followed by arbitrary text that the command can use however it pleases. Currently recognized commands are

```
=head1 Heading Text
=head2 Heading Text
=head3 Heading Text
=head4 Heading Text
=over indentlevel
=item stuff
=back
=cut
=pod
=begin format
=end format
=for format text...
```

To explain them each in detail:

```
=head1 Heading Text
=head2 Heading Text
=head3 Heading Text
=head4 Heading Text
```

Head1 through head4 produce headings, head1 being the highest level. The text in the rest of this paragraph is the content of the heading. For example:

```
=head2 Object Attributes
```

The text "Object Attributes" comprises the heading there. (Note that head3 and head4 are recent additions, not supported in older Pod translators.) The text in these heading commands can use formatting codes, as seen here:

```
=head2 Possible Values for C<$/>
```

Such commands are explained in the "[Formatting Codes in Formatting Codes](#)" section, below.

```
=over indentlevel
=item stuff...
=back
```

Item, over, and back require a little more explanation: "=over" starts a region specifically for the generation of a list using "=item" commands, or for indenting (groups of) normal paragraphs. At the end of your list, use "=back" to end it. The *indentlevel* option to "=over" indicates how far over to indent, generally in ems (where one em is the width of an "M" in the document's base font) or roughly comparable units; if there is no *indentlevel* option, it defaults to four. (And some formatters may just ignore whatever *indentlevel* you provide.) In the *stuff* in `=item stuff...`, you may use formatting codes, as seen here:

```
=item Using C<$|> to Control Buffering
```

Such commands are explained in the "[Formatting Codes in Formatting Codes](#)" section, below.

Note also that there are some basic rules to using "=over" ... "=back" regions:

- Don't use "=item"s outside of an "=over" ... "=back" region.
- The first thing after the "=over" command should be an "=item", unless there aren't going to be any items at all in this "=over" ... "=back" region.
- Don't put "=head*n*" commands inside an "=over" ... "=back" region.
- And perhaps most importantly, keep the items consistent: either use "=item *" for all of them, to produce bullets; or use "=item 1.", "=item 2.", etc., to produce numbered lists; or use "=item foo", "=item bar", etc. — namely, things that look nothing like bullets or numbers.

If you start with bullets or numbers, stick with them, as formatters use the first "=item" type to decide how to format the list.

```
=cut
```

To end a Pod block, use a blank line, then a line beginning with "=cut", and a blank line after it. This lets Perl (and the Pod formatter) know that this is where Perl code is resuming. (The blank line before the "=cut" is not technically necessary, but many older Pod processors require it.)

```
=pod
```

The "=pod" command by itself doesn't do much of anything, but it signals to Perl (and Pod formatters) that a Pod block starts here. A Pod block starts with *any* command paragraph, so a "=pod" command is usually used just when you want to start a Pod block with an ordinary paragraph or a verbatim paragraph. For example:

```
=item stuff()

This function does stuff.

=cut

sub stuff {
    ...
}

=pod
```

Remember to check its return value, as in:

```
stuff() || die "Couldn't do stuff!";

=cut

=begin formatname
=end formatname
=for formatname text...
```

For, begin, and end will let you have regions of text/code/data that are not generally interpreted as normal Pod text, but are passed directly to particular formatters, or are otherwise special. A formatter that can use that format will use the region, otherwise it will be completely ignored.

A command "`=begin formatname`", some paragraphs, and a command "`=end formatname`", mean that the text/data inbetween is meant for formatters that understand the special format called *formatname*. For example,

```
=begin html

<hr> 
<p> This is a raw HTML paragraph </p>

=end html
```

The command "`=for formatname text...`" specifies that the remainder of just this paragraph (starting right after *formatname*) is in that special format.

```
=for html <hr> 
<p> This is a raw HTML paragraph </p>
```

This means the same thing as the above "`=begin html`" ... "`=end html`" region.

That is, with "`=for`", you can have only one paragraph's worth of text (i.e., the text in "`=foo targetname text...`"), but with "`=begin targetname`" ... "`=end targetname`", you can have any amount of stuff inbetween. (Note that there still must be a blank line after the "`=begin`" command and a blank line before the "`=end`" command.)

Here are some examples of how to use these:

```
=begin html

<br>Figure 1.<br><IMG SRC="figure1.png"><br>

=end html

=begin text

-----
|   foo       |
|               |
|               | bar |
|               |
|               |
-----

^^^^ Figure 1. ^^^^

=end text
```

Some format names that formatters currently are known to accept include "roff", "man", "latex", "tex", "text", and "html". (Some formatters will treat some of these as synonyms.)

A format name of "comment" is common for just making notes (presumably to yourself) that won't appear in any formatted version of the Pod document:

```
=for comment
Make sure that all the available options are documented!
```


Some *formatnames* will require a leading colon (as in "`=for :formatname`", or "`=begin :formatname`" ... "`=end :formatname`"), to signal that the text is not raw data, but instead is Pod text (i.e., possibly containing formatting codes) that's just not for normal formatting (e.g., may not be a normal-use paragraph, but might be for formatting as a footnote).

And don't forget, when using any command, that the command lasts up until the end of its *paragraph*, not its line. So in the examples below, you can see that every command needs the blank line after it, to end its paragraph.

Some examples of lists include:

```
=over

=item *

First item

=item *

Second item

=back

=over

=item Foo()

Description of Foo function

=item Bar()

Description of Bar function

=back
```

Formatting Codes

In ordinary paragraphs and in some command paragraphs, various formatting codes (a.k.a. "interior sequences") can be used:

`=for comment`

"interior sequences" is such an opaque term.

Prefer "formatting codes" instead.

`I<text>` — **italic text**

Used for emphasis ("`be I<careful!>`") and parameters ("`redo I<LABEL>`").

`B<text>` — **bold text**

Used for switches ("`perl's B<-n> switch`"), programs ("`some systems provide a B<chfn> for that`"), emphasis ("`be B<careful!>`"), and so on ("`and that feature is known as B<autovivification>`").

`C<code>` — **code text**

Renders code in a typewriter font, or gives some other indication that this represents program text ("`C<gmtime($^T)>`") or some other form of computerese ("`C<drwxr-xr-x>`").

`L<name>` — **a hyperlink**

There are various syntaxes, listed below. In the syntaxes given, `text`, `name`, and `section` cannot contain the characters `'` and `|`; and any `'<` or `'>` should be matched.

- `L<name>`

Link to a Perl manual page (e.g., `L<Net::Ping>`). Note that `name` should not contain spaces. This syntax is also occasionally used for references to UNIX man pages, as in `L<crontab(5)>`.

- `L<name/"sec">` or `L<name/sec>`
Link to a section in other manual page. E.g., `L<perlsyn/"For Loops">`
- `L</"sec">` or `L</sec>` or `L<"sec">`
Link to a section in this manual page. E.g., `L</"Object Methods">`

A section is started by the named heading or item. For example, `L<perlvar/$.>` or `L<perlvar/"$.">` both link to the section started by `"=item $."` in `perlvar`. And `L<perlsyn/For Loops>` or `L<perlsyn/"For Loops">` both link to the section started by `"=head2 For Loops"` in `perlsyn`.

To control what text is used for display, you use `"L<text | ...>"`, as in:

- `L<text | name>`
Link this text to that manual page. E.g., `L<Perl Error Messages|perldiag>`
- `L<text | name/"sec">` or `L<text | name/sec>`
Link this text to that section in that manual page. E.g., `L<SWITCH statements|perlsyn/"Basic BLOCKs and Switch Statements">`
- `L<text | /"sec">` or `L<text | /sec>` or `L<text | "sec">`
Link this text to that section in this manual page. E.g., `L<the various attributes|/"Member Data">`

Or you can link to a web page:

- `L<scheme:...>`
Links to an absolute URL. For example, `L<http://www.perl.org/>`. But note that there is no corresponding `L<text | scheme:...>` syntax, for various reasons.

E<escape> — a character escape

Very similar to HTML/XML `&foo`; "entity references":

- `E<lt>` — a literal `<` (less than)
- `E<gt>` — a literal `>` (greater than)
- `E<verbar>` — a literal `|` (*vertical bar*)
- `E<sol>` = a literal `/` (*solidus*)

The above four are optional except in other formatting codes, notably `L<...>`, and when preceded by a capital letter.

- `E<htmlname>`
Some non-numeric HTML entity name, such as `E<eacute>`, meaning the same thing as `é`; in HTML — i.e., a lowercase e with an acute (/–shaped) accent.
- `E<number>`

The ASCII/Latin-1/Unicode character with that number. A leading `"0x"` means that *number* is hex, as in `E<0x201E>`. A leading `"0"` means that *number* is octal, as in `E<075>`. Otherwise *number* is interpreted as being in decimal, as in `E<181>`.

Note that older Pod formatters might not recognize octal or hex numeric escapes, and that many formatters cannot reliably render characters above 255. (Some formatters may even have to use compromised renderings of Latin-1 characters, like rendering `E<eacute>` as just a plain "e".)

F<filename> — used for filenames

Typically displayed in italics. Example: "F<.cshrc>"

S<text> — text contains non-breaking spaces

This means that the words in *text* should not be broken across lines. Example: S<\$x ? \$y : \$z>.

X<topic name> — an index entry

This is ignored by most formatters, but some may use it for building indexes. It always renders as empty-string. Example: X<absolutizing relative URLs>

Z<> — a null (zero-effect) formatting code

This is rarely used. It's one way to get around using an E<...> code sometimes. For example, instead of "NE<lt>3" (for "N<3") you could write "NZ<><3" (the "Z<>" breaks up the "N" and the "<" so they can't be considered the part of a (fictitious) "N<...>" code.

=for comment

This was formerly explained as a "zero-width character". But it in most parser models, it parses to nothing at all, as opposed to parsing as if it were a `or`, which are REAL zero-width characters.

So "width" and "character" are exactly the wrong words.

Most of the time, you will need only a single set of angle brackets to delimit the beginning and end of formatting codes. However, sometimes you will want to put a real right angle bracket (a greater-than sign, `>`) inside of a formatting code. This is particularly common when using a formatting code to provide a different font-type for a snippet of code. As with all things in Perl, there is more than one way to do it. One way is to simply escape the closing bracket using an E code:

```
C<$a E<lt>=E<gt> $b>
```

This will produce: "\$a <=> \$b"

A more readable, and perhaps more "plain" way is to use an alternate set of delimiters that doesn't require a single `"` to be escaped. With the Pod formatters that are standard starting with perl5.5.660, doubled angle brackets ("`<<`" and `>>`") may be used *if and only if there is whitespace right after the opening delimiter and whitespace right before the closing delimiter!* For example, the following will do the trick:

```
C<< $a <=> $b >>
```

In fact, you can use as many repeated angle-brackets as you like so long as you have the same number of them in the opening and closing delimiters, and make sure that whitespace immediately follows the last `<` of the opening delimiter, and immediately precedes the first `>` of the closing delimiter. (The whitespace is ignored.) So the following will also work:

```
C<<< $a <=> $b >>>
C<<<< $a <=> $b >>>>
```

And they all mean exactly the same as this:

```
C<$a E<lt>=E<gt> $b>
```

As a further example, this means that if you wanted to put these bits of code in C (code) style:

```
open(X, ">>thing.dat") || die $!
$foo->bar();
```

you could do it like so:

```
C<<< open(X, ">>thing.dat") || die $! >>>
C<< $foo->bar(); >>
```

which is presumably easier to read than the old way:

```
C<open(X, "E<gt>E<gt>thing.dat") || die $!>
C<$foo-E<gt>bar(); >>
```

This is currently supported by `pod2text` (`Pod::Text`), `pod2man` (`Pod::Man`), and any other `pod2xxx` or `Pod::Xxxx` translators that use `Pod::Parser` 1.093 or later, or `Pod::Tree` 1.02 or later.

The Intent

The intent is simplicity of use, not power of expression. Paragraphs look like paragraphs (block format), so that they stand out visually, and so that I could run them through `fmt` easily to reformat them (that's F7 in my version of **vi**, or Esc Q in my version of **emacs**). I wanted the translator to always leave the ' and ' and " quotes alone, in verbatim mode, so I could slurp in a working program, shift it over four spaces, and have it print out, er, verbatim. And presumably in a monospace font.

The Pod format is not necessarily sufficient for writing a book. Pod is just meant to be an idiot-proof common source for `nroff`, HTML, TeX, and other markup languages, as used for online documentation. Translators exist for **pod2text**, **pod2html**, **pod2man** (that's for `nroff(1)` and `troff(1)`), **pod2latex**, and **pod2fm**. Various others are available in CPAN.

Embedding Pods in Perl Modules

You can embed Pod documentation in your Perl modules and scripts. Start your documentation with an empty line, a `=head1` command at the beginning, and end it with a `=cut` command and an empty line. Perl will ignore the Pod text. See any of the supplied library modules for examples. If you're going to put your Pod at the end of the file, and you're using an `__END__` or `__DATA__` cut mark, make sure to put an empty line there before the first Pod command.

```
__END__
```

```
=head1 NAME
```

```
Time::Local - efficiently compute time from local and GMT time
```

Without that empty line before the `=head1`, many translators wouldn't have recognized the `=head1` as starting a Pod block.

Hints for Writing Pod

- The **podchecker** command is provided for checking Pod syntax for errors and warnings. For example, it checks for completely blank lines in Pod blocks and for unknown commands and formatting codes. You should still also pass your document through one or more translators and proofread the result, or print out the result and proofread that. Some of the problems found may be bugs in the translators, which you may or may not wish to work around.
- If you're more familiar with writing in HTML than with writing in Pod, you can try your hand at writing documentation in simple HTML, and converting it to Pod with the experimental [Pod::HTML2Pod/Pod::HTML2Pod](#) module, (available in CPAN), and looking at the resulting code. The experimental [Pod::PXML/Pod::PXML](#) module in CPAN might also be useful.
- Many older Pod translators require the lines before every Pod command and after every Pod command (including `=cut`!) to be a blank line. Having something like this:

```
# - - - - -
=item $firecracker->boom()

This noisily detonates the firecracker object.
=cut
sub boom {
    ...
```

...will make such Pod translators completely fail to see the Pod block at all.

Instead, have it like this:

```
# - - - - -  
=item $firecracker->boom()  
  
This noisily detonates the firecracker object.  
  
=cut  
  
sub boom {  
    ...  
}
```

- Some older Pod translators require paragraphs (including command paragraphs like "`=head2 Functions`") to be separated by *completely* empty lines. If you have an apparently empty line with some spaces on it, this might not count as a separator for those translators, and that could cause odd formatting.
- Older translators might add wording around an `L<>` link, so that `L<Foo::Bar>` may become "the Foo::Bar manpage", for example. So you shouldn't write things like the `L<foo>` documentation, if you want the translated document to read sensibly — instead write the `L<Foo::Bar|Foo::Bar>` documentation or `L<the Foo::Bar documentation|Foo::Bar>`, to control how the link comes out.
- Going past the 70th column in a verbatim block might be ungracefully wrapped by some formatters.

SEE ALSO

[perlpodspec](#), *PODs: Embedded Documentation in perlsyn*, [perlnewmod](#), [perl doc](#), [pod2html](#), [pod2man](#), [podchecker](#).

AUTHOR

Larry Wall, Sean M. Burke

NAME

perlpodspec – Plain Old Documentation: format specification and notes

DESCRIPTION

This document is detailed notes on the Pod markup language. Most people will only have to read [perlpod/perlpod](#) to know how to write in Pod, but this document may answer some incidental questions to do with parsing and rendering Pod.

In this document, "must" / "must not", "should" / "should not", and "may" have their conventional (cf. RFC 2119) meanings: "X must do Y" means that if X doesn't do Y, it's against this specification, and should really be fixed. "X should do Y" means that it's recommended, but X may fail to do Y, if there's a good reason. "X may do Y" is merely a note that X can do Y at will (although it is up to the reader to detect any connotation of "and I think it would be *nice* if X did Y" versus "it wouldn't really *bother* me if X did Y").

Notably, when I say "the parser should do Y", the parser may fail to do Y, if the calling application explicitly requests that the parser *not* do Y. I often phrase this as "the parser should, by default, do Y." This doesn't *require* the parser to provide an option for turning off whatever feature Y is (like expanding tabs in verbatim paragraphs), although it implicates that such an option *may* be provided.

Pod Definitions

Pod is embedded in files, typically Perl source files — although you can write a file that's nothing but Pod.

A **line** in a file consists of zero or more non-newline characters, terminated by either a newline or the end of the file.

A **newline sequence** is usually a platform-dependent concept, but Pod parsers should understand it to mean any of CR (ASCII 13), LF (ASCII 10), or a CRLF (ASCII 13 followed immediately by ASCII 10), in addition to any other system-specific meaning. The first CR/CRLF/LF sequence in the file may be used as the basis for identifying the newline sequence for parsing the rest of the file.

A **blank line** is a line consisting entirely of zero or more spaces (ASCII 32) or tabs (ASCII 9), and terminated by a newline or end-of-file. A **non-blank line** is a line containing one or more characters other than space or tab (and terminated by a newline or end-of-file).

(*Note:* Many older Pod parsers did not accept a line consisting of spaces/tabs and then a newline as a blank line — the only lines they considered blank were lines consisting of *no characters at all*, terminated by a newline.)

Whitespace is used in this document as a blanket term for spaces, tabs, and newline sequences. (By itself, this term usually refers to literal whitespace. That is, sequences of whitespace characters in Pod source, as opposed to "E<32", which is a formatting code that *denotes* a whitespace character.)

A **Pod parser** is a module meant for parsing Pod (regardless of whether this involves calling callbacks or building a parse tree or directly formatting it). A **Pod formatter** (or **Pod translator**) is a module or program that converts Pod to some other format (HTML, plaintext, TeX, PostScript, RTF). A **Pod processor** might be a formatter or translator, or might be a program that does something else with the Pod (like wordcounting it, scanning for index points, etc.).

Pod content is contained in **Pod blocks**. A Pod block starts with a line that matches `<m/A=[a-zA-Z]/`, and continues up to the next line that matches `m/\A=cut/` — or up to the end of the file, if there is no `m/\A=cut/` line.

=for comment

The current perlsyn says:

[beginquote]

Note that pod translators should look at only paragraphs beginning with a pod directive (it makes parsing easier), whereas the compiler actually knows to look for pod escapes even in the middle of a paragraph. This means that the following secret stuff will be ignored

by both the compiler and the translators.

```
$a=3;
=secret stuff
warn "Neither POD nor CODE!?"
=cut back
print "got $a\n";
```

You probably shouldn't rely upon the `warn()` being podded out forever.

Not all pod translators are well-behaved in this regard, and perhaps the compiler will become pickier.

[endquote]

I think that those paragraphs should just be removed; paragraph-based parsing seems to have been largely abandoned, because of the hassle with non-empty blank lines messing up what people meant by "paragraph". Even if the "it makes parsing easier" bit were especially true, it wouldn't be worth the confusion of having perl and pod2whatever actually disagree on what can constitute a Pod block.

Within a Pod block, there are **Pod paragraphs**. A Pod paragraph consists of non-blank lines of text, separated by one or more blank lines.

For purposes of Pod processing, there are four types of paragraphs in a Pod block:

- A **command paragraph** (also called a "directive"). The first line of this paragraph must match `m/\A= [a-zA-Z] /`. Command paragraphs are typically one line, as in:

```
=head1 NOTES

=item *
```

But they may span several (non-blank) lines:

```
=for comment
Hm, I wonder what it would look like if
you tried to write a BNF for Pod from this.

=head3 Dr. Strangelove, or: How I Learned to
Stop Worrying and Love the Bomb
```

Some command paragraphs allow formatting codes in their content (i.e., after the part that matches `m/\A= [a-zA-Z] \S*\s*/`), as in:

```
=head1 Did You Remember to C<use strict;>?
```

In other words, the Pod processing handler for "head1" will apply the same processing to "Did You Remember to C<use strict;?" that it would to an ordinary paragraph — i.e., formatting codes (like "C<...") are parsed and presumably formatted appropriately, and whitespace in the form of literal spaces and/or tabs is not significant.

- A **verbatim paragraph**. The first line of this paragraph must be a literal space or tab, and this paragraph must not be inside a "`=begin identifier`", ... "`=end identifier`" sequence unless "`identifier`" begins with a colon (":"). That is, if a paragraph starts with a literal space or tab, but *is* inside a "`=begin identifier`", ... "`=end identifier`" region, then it's a data paragraph, unless "`identifier`" begins with a colon.

Whitespace *is* significant in verbatim paragraphs (although, in processing, tabs are probably expanded).

- An **ordinary paragraph**. A paragraph is an ordinary paragraph if its first line matches neither `m/\A= [a-zA-Z] /` nor `m/\A[\t] /`, and if it's not inside a "`=begin identifier`", ... "`=end identifier`" sequence unless "`identifier`" begins with a colon (":").

- **A data paragraph.** This is a paragraph that *is* inside a "`=begin identifier`" ... "`=end identifier`" sequence where "*identifier*" does *not* begin with a literal colon (":"). In some sense, a data paragraph is not part of Pod at all (i.e., effectively it's "out-of-band"), since it's not subject to most kinds of Pod parsing; but it is specified here, since Pod parsers need to be able to call an event for it, or store it in some form in a parse tree, or at least just parse *around* it.

For example: consider the following paragraphs:

```
# <- that's the 0th column

=head1 Foo

Stuff

$foo->bar

=cut
```

Here, "`=head1 Foo`" and "`=cut`" are command paragraphs because the first line of each matches `m/\A= [a-zA-Z] /. "[space][space]$foo->bar"` is a verbatim paragraph, because its first line starts with a literal whitespace character (and there's no "`=begin`"..."`=end`" region around).

The "`=begin identifier`" ... "`=end identifier`" commands stop paragraphs that they surround from being parsed as data or verbatim paragraphs, if *identifier* doesn't begin with a colon. This is discussed in detail in the section [/About Data Paragraphs and "=begin=end" Regions](#).

Pod Commands

This section is intended to supplement and clarify the discussion in [Command Paragraph in *perlpod*](#). These are the currently recognized Pod commands:

`"=head1", "=head2", "=head3", "=head4"`

This command indicates that the text in the remainder of the paragraph is a heading. That text may contain formatting codes. Examples:

```
=head1 Object Attributes

=head3 What B<Not> to Do!
```

`"=pod"`

This command indicates that this paragraph begins a Pod block. (If we are already in the middle of a Pod block, this command has no effect at all.) If there is any text in this command paragraph after "`=pod`", it must be ignored. Examples:

```
=pod

This is a plain Pod paragraph.

=pod This text is ignored.
```

`"=cut"`

This command indicates that this line is the end of this previously started Pod block. If there is any text after "`=cut`" on the line, it must be ignored. Examples:

```
=cut

=cut The documentation ends here.

=cut
# This is the first line of program text.
sub foo { # This is the second.
```

It is an error to try to *start* a Pod block with a "`=cut`" command. In that case, the Pod processor must halt parsing of the input file, and must by default emit a warning.

"=over"

This command indicates that this is the start of a list/indent region. If there is any text following the "=over", it must consist of only a nonzero positive numeral. The semantics of this numeral is explained in the [section](#), further below. Formatting codes are not expanded. Examples:

```
=over 3

=over 3.5

=over
```

"=item"

This command indicates that an item in a list begins here. Formatting codes are processed. The semantics of the (optional) text in the remainder of this paragraph are explained in the [section](#), further below. Examples:

```
=item

=item *

=item      *

=item 14

=item 3.

=item C<< $thing->stuff(I<dodad>) >>

=item For transporting us beyond seas to be tried for pretended
offenses

=item He is at this time transporting large armies of foreign
mercenaries to complete the works of death, desolation and
tyranny, already begun with circumstances of cruelty and perfidy
scarcely paralleled in the most barbarous ages, and totally
unworthy the head of a civilized nation.
```

"=back"

This command indicates that this is the end of the region begun by the most recent "=over" command. It permits no text after the "=back" command.

"=begin formatname"

This marks the following paragraphs (until the matching "=end formatname") as being for some special kind of processing. Unless "formatname" begins with a colon, the contained non-command paragraphs are data paragraphs. But if "formatname" *does* begin with a colon, then non-command paragraphs are ordinary paragraphs or data paragraphs. This is discussed in detail in the [section *About Data Paragraphs and "=begin=end" Regions*](#).

It is advised that formatnames match the regexp `m/\A:?[-a-zA-Z0-9_]+\z/`. Implementors should anticipate future expansion in the semantics and syntax of the first parameter to "=begin"/"=end"/"=for".

"=end formatname"

This marks the end of the region opened by the matching "=begin formatname" region. If "formatname" is not the formatname of the most recent open "=begin formatname" region, then this is an error, and must generate an error message. This is discussed in detail in the [section *About Data Paragraphs and "=begin=end" Regions*](#).

"=for formatname text..."

This is synonymous with:

```
=begin formatname
text...
=end formatname
```

That is, it creates a region consisting of a single paragraph; that paragraph is to be treated as a normal paragraph if "formatname" begins with a ":"; if "formatname" *doesn't* begin with a colon, then "text..." will constitute a data paragraph. There is no way to use "=for formatname text..." to express "text..." as a verbatim paragraph.

If a Pod processor sees any command other than the ones listed above (like "=head", or "=head1", or "=stuff", or "=cuttlefish", or "=w123"), that processor must by default treat this as an error. It must not process the paragraph beginning with that command, must by default warn of this as an error, and may abort the parse. A Pod parser may allow a way for particular applications to add to the above list of known commands, and to stipulate, for each additional command, whether formatting codes should be processed.

Future versions of this specification may add additional commands.

Pod Formatting Codes

(Note that in previous drafts of this document and of perlpod, formatting codes were referred to as "interior sequences", and this term may still be found in the documentation for Pod parsers, and in error messages from Pod processors.)

There are two syntaxes for formatting codes:

- A formatting code starts with a capital letter (just US-ASCII [A-Z]) followed by a "<", any number of characters, and ending with the first matching "'". Examples:

```
That's what I<you> think!
```

```
What's C<dump()> for?
```

```
X<C<chmod> and C<unlink()> Under Different Operating Systems>
```

- A formatting code starts with a capital letter (just US-ASCII [A-Z]) followed by two or more "<"s, one or more whitespace characters, any number of characters, one or more whitespace characters, and ending with the first matching sequence of two or more "'s, where the number of "'s equals the number of "<"s in the opening of this formatting code. Examples:

```
That's what I<< you >> think!
```

```
C<<< open(X, ">>thing.dat") || die $! >>>
```

```
B<< $foo->bar(); >>
```

With this syntax, the whitespace character(s) after the "C<<<" and before the "'" (or whatever letter) are *not* renderable — they do not signify whitespace, are merely part of the formatting codes themselves. That is, these are all synonymous:

```
C<thing>
C<< thing >>
C<<      thing      >>
C<<<  thing >>>
C<<<<
thing
      >>>>
```

and so on.

In parsing Pod, a notably tricky part is the correct parsing of (potentially nested!) formatting codes. Implementors should consult the code in the `parse_text` routine in `Pod::Parser` as an example of a correct implementation.

I<text> — italic text

See the brief discussion in *Formatting Codes in perlpod*.

B<text> — bold text

See the brief discussion in *Formatting Codes in perlpod*.

C<code> — code text

See the brief discussion in *Formatting Codes in perlpod*.

F<filename> — style for filenames

See the brief discussion in *Formatting Codes in perlpod*.

X<topic name> — an index entry

See the brief discussion in *Formatting Codes in perlpod*.

This code is unusual in that most formatters completely discard this code and its content. Other formatters will render it with invisible codes that can be used in building an index of the current document.

Z<> — a null (zero-effect) formatting code

Discussed briefly in *Formatting Codes in perlpod*.

This code is unusual is that it should have no content. That is, a processor may complain if it sees Z<potatoes>. Whether or not it complains, the *potatoes* text should ignored.

L<name> — a hyperlink

The complicated syntaxes of this code are discussed at length in *Formatting Codes in perlpod*, and implementation details are discussed below, in *"About L<...> Codes"*. Parsing the contents of L<content is tricky. Notably, the content has to be checked for whether it looks like a URL, or whether it has to be split on literal "|" and/or "/" (in the right order!), and so on, *before* E<... codes are resolved.

E<escape> — a character escape

See *Formatting Codes in perlpod*, and several points in */Notes on Implementing Pod Processors*.

S<text> — text contains non-breaking spaces

This formatting code is syntactically simple, but semantically complex. What it means is that each space in the printable content of this code signifies a nonbreaking space.

Consider:

```
C<$x ? $y      :  $z>
```

```
S<C<$x ? $y      :  $z>>
```

Both signify the monospace (c[ode] style) text consisting of "\$x", one space, "?", one space, ":", one space, "\$z". The difference is that in the latter, with the S code, those spaces are not "normal" spaces, but instead are nonbreaking spaces.

If a Pod processor sees any formatting code other than the ones listed above (as in "N<...", or "Q<...", etc.), that processor must by default treat this as an error. A Pod parser may allow a way for particular applications to add to the above list of known formatting codes; a Pod parser might even allow a way to stipulate, for each additional command, whether it requires some form of special processing, as L<... does.

Future versions of this specification may add additional formatting codes.

Historical note: A few older Pod processors would not see a "" as closing a "C<" code, if the "" was immediately preceded by a "-". This was so that this:

```
C<$foo->bar>
```

would parse as equivalent to this:

```
C<$foo-E<lt>bar>
```

instead of as equivalent to a "C" formatting code containing only "\$foo-", and then a "bar" outside the "C" formatting code. This problem has since been solved by the addition of syntaxes like this:

```
C<< $foo->bar >>
```

Compliant parsers must not treat "-" as special.

Formatting codes absolutely cannot span paragraphs. If a code is opened in one paragraph, and no closing code is found by the end of that paragraph, the Pod parser must close that formatting code, and should complain (as in "Unterminated I code in the paragraph starting at line 123: 'Time objects are not...'"). So these two paragraphs:

```
I<I told you not to do this!
```

```
Don't make me say it again!>
```

...must *not* be parsed as two paragraphs in italics (with the I code starting in one paragraph and starting in another.) Instead, the first paragraph should generate a warning, but that aside, the above code must parse as if it were:

```
I<I told you not to do this!>
```

```
Don't make me say it again!E<gt>
```

(In SGMLish jargon, all Pod commands are like block-level elements, whereas all Pod formatting codes are like inline-level elements.)

Notes on Implementing Pod Processors

The following is a long section of miscellaneous requirements and suggestions to do with Pod processing.

- Pod formatters should tolerate lines in verbatim blocks that are of any length, even if that means having to break them (possibly several times, for very long lines) to avoid text running off the side of the page. Pod formatters may warn of such line-breaking. Such warnings are particularly appropriate for lines are over 100 characters long, which are usually not intentional.
- Pod parsers must recognize *all* of the three well-known newline formats: CR, LF, and CRLF. See [perlport/perlport](#).
- Pod parsers should accept input lines that are of any length.
- Since Perl recognizes a Unicode Byte Order Mark at the start of files as signaling that the file is Unicode encoded as in UTF-16 (whether big-endian or little-endian) or UTF-8, Pod parsers should do the same. Otherwise, the character encoding should be understood as being UTF-8 if the first highbit byte sequence in the file seems valid as a UTF-8 sequence, or otherwise as Latin-1.

Future versions of this specification may specify how Pod can accept other encodings. Presumably treatment of other encodings in Pod parsing would be as in XML parsing: whatever the encoding declared by a particular Pod file, content is to be stored in memory as Unicode characters.

- The well known Unicode Byte Order Marks are as follows: if the file begins with the two literal byte values 0xFE 0xFF, this is the BOM for big-endian UTF-16. If the file begins with the two literal byte value 0xFF 0xFE, this is the BOM for little-endian UTF-16. If the file begins with the three literal byte values 0xEF 0xBB 0xBF, this is the BOM for UTF-8.

```
=for comment
```

```
use bytes; print map sprintf(" 0x%02X", ord $_) , split ' ', "\x{feff}";
0xEF 0xBB 0xBF
```

```
=for comment
```

```
If toke.c is modified to support UTF32, add mention of those here.
```

- A naive but sufficient heuristic for testing the first highbit byte–sequence in a BOM–less file (whether in code or in Pod!), to see whether that sequence is valid as UTF–8 (RFC 2279) is to check whether that the first byte in the sequence is in the range 0xC0 – 0xFD *and* whether the next byte is in the range 0x80 – 0xBF. If so, the parser may conclude that this file is in UTF–8, and all highbit sequences in the file should be assumed to be UTF–8. Otherwise the parser should treat the file as being in Latin–1. In the unlikely circumstance that the first highbit sequence in a truly non–UTF–8 file happens to appear to be UTF–8, one can cater to our heuristic (as well as any more intelligent heuristic) by prefacing that line with a comment line containing a highbit sequence that is clearly *not* valid as UTF–8. A line consisting of simply "#", an e–acute, and any non–highbit byte, is sufficient to establish this file’s encoding.

=for comment

If/WHEN some brave soul makes these heuristics into a generic text–file class (or PerlIO layer?), we can presumably delete mention of these icky details from this file, and can instead tell people to just use appropriate class/layer. Auto–recognition of newline sequences would be another desirable feature of such a class/layer.
HINT HINT HINT.

=for comment

"The probability that a string of characters in any other encoding appears as valid UTF–8 is low" – RFC2279

- This document’s requirements and suggestions about encodings do not apply to Pod processors running on non–ASCII platforms, notably EBCDIC platforms.
- Pod processors must treat a "=for [label] [content...]" paragraph as meaning the same thing as a "=begin [label]" paragraph, content, and an "=end [label]" paragraph. (The parser may conflate these two constructs, or may leave them distinct, in the expectation that the formatter will nevertheless treat them the same.)
- When rendering Pod to a format that allows comments (i.e., to nearly any format other than plaintext), a Pod formatter must insert comment text identifying its name and version number, and the name and version numbers of any modules it might be using to process the Pod. Minimal examples:

```
%% POD::Pod2PS v3.14159, using POD::Parser v1.92
<!-- Pod::HTML v3.14159, using POD::Parser v1.92 -->
{\doccomm generated by Pod::Tree::RTF 3.14159 using Pod::Tree 1.08}
.\ " Pod::Man version 3.14159, using POD::Parser version 1.92
```

Formatters may also insert additional comments, including: the release date of the Pod formatter program, the contact address for the author(s) of the formatter, the current time, the name of input file, the formatting options in effect, version of Perl used, etc.

Formatters may also choose to note errors/warnings as comments, besides or instead of emitting them otherwise (as in messages to STDERR, or dieing).

- Pod parsers *may* emit warnings or error messages ("Unknown E code E<zslig!") to STDERR (whether through printing to STDERR, or warning/carping, or dieing/croaking), but *must* allow suppressing all such STDERR output, and instead allow an option for reporting errors/warnings in some other way, whether by triggering a callback, or noting errors in some attribute of the document object, or some similarly unobtrusive mechanism — or even by appending a "Pod Errors" section to the end of the parsed form of the document.

- In cases of exceptionally aberrant documents, Pod parsers may abort the parse. Even then, using `dieing/croaking` is to be avoided; where possible, the parser library may simply close the input file and add text like `**** Formatting Aborted ****` to the end of the (partial) in-memory document.
- In paragraphs where formatting codes (like `E<...`, `B<...`) are understood (i.e., *not* verbatim paragraphs, but *including* ordinary paragraphs, and command paragraphs that produce renderable text, like `"=head1"`), literal whitespace should generally be considered "insignificant", in that one literal space has the same meaning as any (nonzero) number of literal spaces, literal newlines, and literal tabs (as long as this produces no blank lines, since those would terminate the paragraph). Pod parsers should compact literal whitespace in each processed paragraph, but may provide an option for overriding this (since some processing tasks do not require it), or may follow additional special rules (for example, specially treating period-space-space or period-newline sequences).
- Pod parsers should not, by default, try to coerce apostrophe (') and quote (") into smart quotes (little 9's, 66's, 99's, etc), nor try to turn backtick (`) into anything else but a single backtick character (distinct from an openquote character!), nor `"--"` into anything but two minus signs. They *must never* do any of those things to text in `C<...` formatting codes, and never *ever* to text in verbatim paragraphs.
- When rendering Pod to a format that has two kinds of hyphens (–), one that's a nonbreaking hyphen, and another that's a breakable hyphen (as in "object-oriented", which can be split across lines as "object–", newline, "oriented"), formatters are encouraged to generally translate `"–"` to nonbreaking hyphen, but may apply heuristics to convert some of these to breaking hyphens.
- Pod formatters should make reasonable efforts to keep words of Perl code from being broken across lines. For example, `"Foo::Bar"` in some formatting systems is seen as eligible for being broken across lines as `"Foo::"` newline `"Bar"` or even `"Foo::–"` newline `"Bar"`. This should be avoided where possible, either by disabling all line-breaking in mid-word, or by wrapping particular words with internal punctuation in "don't break this across lines" codes (which in some formats may not be a single code, but might be a matter of inserting non-breaking zero-width spaces between every pair of characters in a word.)
- Pod parsers should, by default, expand tabs in verbatim paragraphs as they are processed, before passing them to the formatter or other processor. Parsers may also allow an option for overriding this.
- Pod parsers should, by default, remove newlines from the end of ordinary and verbatim paragraphs before passing them to the formatter. For example, while the paragraph you're reading now could be considered, in Pod source, to end with (and contain) the newline(s) that end it, it should be processed as ending with (and containing) the period character that ends this sentence.
- Pod parsers, when reporting errors, should make some effort to report an approximate line number ("Nested E<'s in Paragraph #52, near line 633 of Thing/Foo.pm!"), instead of merely noting the paragraph number ("Nested E<'s in Paragraph #52 of Thing/Foo.pm!"). Where this is problematic, the paragraph number should at least be accompanied by an excerpt from the paragraph ("Nested E<'s in Paragraph #52 of Thing/Foo.pm, which begins 'Read/write accessor for the C<interest rate attribute...'").
- Pod parsers, when processing a series of verbatim paragraphs one after another, should consider them to be one large verbatim paragraph that happens to contain blank lines. I.e., these two lines, which have a blank line between them:

```
use Foo;

print Foo->VERSION
```

should be unified into one paragraph (`"\tuse Foo;\n\n\tprint Foo-VERSION"`) before being passed to the formatter or other processor. Parsers may also allow an option for overriding this.

While this might be too cumbersome to implement in event-based Pod parsers, it is straightforward for parsers that return parse trees.

- Pod formatters, where feasible, are advised to avoid splitting short verbatim paragraphs (under twelve lines, say) across pages.
- Pod parsers must treat a line with only spaces and/or tabs on it as a "blank line" such as separates paragraphs. (Some older parsers recognized only two adjacent newlines as a "blank line" but would not recognize a newline, a space, and a newline, as a blank line. This is noncompliant behavior.)
- Authors of Pod formatters/processors should make every effort to avoid writing their own Pod parser. There are already several in CPAN, with a wide range of interface styles — and one of them, Pod::Parser, comes with modern versions of Perl.
- Characters in Pod documents may be conveyed either as literals, or by number in E<n codes, or by an equivalent mnemonic, as in E<eacute which is exactly equivalent to E<233.

Characters in the range 32–126 refer to those well known US–ASCII characters (also defined there by Unicode, with the same meaning), which all Pod formatters must render faithfully. Characters in the ranges 0–31 and 127–159 should not be used (neither as literals, nor as E<number codes), except for the literal byte–sequences for newline (13, 13 10, or 10), and tab (9).

Characters in the range 160–255 refer to Latin–1 characters (also defined there by Unicode, with the same meaning). Characters above 255 should be understood to refer to Unicode characters.

- Be warned that some formatters cannot reliably render characters outside 32–126; and many are able to handle 32–126 and 160–255, but nothing above 255.
- Besides the well-known "E<lt" and "E<gt" codes for less–than and greater–than, Pod parsers must understand "E<sol" for "/" (solidus, slash), and "E<verbar" for "|" (vertical bar, pipe). Pod parsers should also understand "E<lchevron" and "E<rchevron" as legacy codes for characters 171 and 187, i.e., "left–pointing double angle quotation mark" = "left pointing guillemet" and "right–pointing double angle quotation mark" = "right pointing guillemet". (These look like little "<<" and ">>", and they are now preferably expressed with the HTML/XHTML codes "E<laquo" and "E<raquo".)
- Pod parsers should understand all "E<html" codes as defined in the entity declarations in the most recent XHTML specification at www.w3.org. Pod parsers must understand at least the entities that define characters in the range 160–255 (Latin–1). Pod parsers, when faced with some unknown "E<identifier" code, shouldn't simply replace it with nullstring (by default, at least), but may pass it through as a string consisting of the literal characters E, less–than, *identifier*, greater–than. Or Pod parsers may offer the alternative option of processing such unknown "E<identifier" codes by firing an event especially for such codes, or by adding a special node–type to the in–memory document tree. Such "E<identifier" may have special meaning to some processors, or some processors may choose to add them to a special error report.
- Pod parsers must also support the XHTML codes "E<quot" for character 34 (doublequote, "), "E<amp" for character 38 (ampersand, &), and "E<apos" for character 39 (apostrophe, ').
- Note that in all cases of "E<whatever", *whatever* (whether an htmlname, or a number in any base) must consist only of alphanumeric characters — that is, *whatever* must watch `m/\A\w+\z/`. So "E< 0 1 2 3 " is invalid, because it contains spaces, which aren't alphanumeric characters. This presumably does not *need* special treatment by a Pod processor; " 0 1 2 3 " doesn't look like a number in any base, so it would presumably be looked up in the table of HTML–like names. Since there isn't (and cannot be) an HTML–like entity called " 0 1 2 3 ", this will be treated as an error. However, Pod processors may treat "E< 0 1 2 3 " or "E<e–acute" as *syntactically* invalid, potentially earning a different error message than the error message (or warning, or event) generated by a merely unknown (but theoretically valid) htmlname, as in "E<qacute" [sic]. However, Pod parsers are not required to make this distinction.
- Note that E<number *must not* be interpreted as simply "codepoint *number* in the current/native character set". It always means only "the character represented by codepoint *number* in Unicode." (This is identical to the semantics of `&#number`; in XML.)

This will likely require many formatters to have tables mapping from treatable Unicode codepoints (such as the "\xE9" for the e–acute character) to the escape sequences or codes necessary for conveying such sequences in the target output format. A converter to *roff would, for example know that "\xE9" (whether conveyed literally, or via an E<... sequence) is to be conveyed as "e*". Similarly, a program rendering Pod in a Mac OS application window, would presumably need to know that "\xE9" maps to codepoint 142 in MacRoman encoding that (at time of writing) is native for Mac OS. Such Unicode2whatever mappings are presumably already widely available for common output formats. (Such mappings may be incomplete! Implementers are not expected to bend over backwards in an attempt to render Cherokee syllabics, Etruscan runes, Byzantine musical symbols, or any of the other weird things that Unicode can encode.) And if a Pod document uses a character not found in such a mapping, the formatter should consider it an unrenderable character.

- If, surprisingly, the implementor of a Pod formatter can't find a satisfactory pre-existing table mapping from Unicode characters to escapes in the target format (e.g., a decent table of Unicode characters to *roff escapes), it will be necessary to build such a table. If you are in this circumstance, you should begin with the characters in the range 0x00A0 – 0x00FF, which is mostly the heavily used accented characters. Then proceed (as patience permits and fastidiousness compels) through the characters that the (X)HTML standards groups judged important enough to merit mnemonics for. These are declared in the (X)HTML specifications at the www.W3.org site. At time of writing (September 2001), the most recent entity declaration files are:

```
http://www.w3.org/TR/xhtml1/DTD/xhtml1-lat1.ent
http://www.w3.org/TR/xhtml1/DTD/xhtml1-special.ent
http://www.w3.org/TR/xhtml1/DTD/xhtml1-symbol.ent
```

Then you can progress through any remaining notable Unicode characters in the range 0x2000–0x204D (consult the character tables at www.unicode.org), and whatever else strikes your fancy. For example, in *xhtml–symbol.ent*, there is the entry:

```
<!ENTITY infin      "&#8734;"> <!-- infinity, U+221E ISOTech -->
```

While the mapping "infin" to the character "\x{221E}" will (hopefully) have been already handled by the Pod parser, the presence of the character in this file means that it's reasonably important enough to include in a formatter's table that maps from notable Unicode characters to the codes necessary for rendering them. So for a Unicode-to-*roff mapping, for example, this would merit the entry:

```
"\x{221E}" => '\(in',
```

It is eagerly hoped that in the future, increasing numbers of formats (and formatters) will support Unicode characters directly (as (X)HTML does with ∞, ∞, or ∞), reducing the need for idiosyncratic mappings of Unicode-to-*my_escapes.

- It is up to individual Pod formatter to display good judgment when confronted with an unrenderable character (which is distinct from an unknown E<thing sequence that the parser couldn't resolve to anything, renderable or not). It is good practice to map Latin letters with diacritics (like "E<eacute"/"E<233") to the corresponding unaccented US–ASCII letters (like a simple character 101, "e"), but clearly this is often not feasible, and an unrenderable character may be represented as "?", or the like. In attempting a sane fallback (as from E<233 to "e"), Pod formatters may use the %Latin1Code_to_fallback table in *Pod::Escapes*/*Pod::Escapes*, or *Text::Unidecode*/*Text::Unidecode*, if available.

For example, this Pod text:

```
magic is enabled if you set C<$Currency> to 'E<euro>'.
```

may be rendered as: "magic is enabled if you set \$Currency to '?'" or as "magic is enabled if you set \$Currency to '[euro]'", or as "magic is enabled if you set \$Currency to '[x20AC]'", etc.

A Pod formatter may also note, in a comment or warning, a list of what unrenderable characters were encountered.

- E<... may freely appear in any formatting code (other than in another E<... or in an Z<). That is, "X<The E<euro1,000,000 Solution" is valid, as is "L<The E<euro1,000,000 Solution|Million::Euros".
- Some Pod formatters output to formats that implement nonbreaking spaces as an individual character (which I'll call "NBSP"), and others output to formats that implement nonbreaking spaces just as spaces wrapped in a "don't break this across lines" code. Note that at the level of Pod, both sorts of codes can occur: Pod can contain a NBSP character (whether as a literal, or as a "E<160" or "E<nbsp" code); and Pod can contain "S<foo I<bar> baz" codes, where "mere spaces" (character 32) in such codes are taken to represent nonbreaking spaces. Pod parsers should consider supporting the optional parsing of "S<foo I<bar> baz" as if it were "fooNBSP I<bar>NBSPbaz", and, going the other way, the optional parsing of groups of words joined by NBSP's as if each group were in a S<... code, so that formatters may use the representation that maps best to what the output format demands.
- Some processors may find that the S<...> code is easiest to implement by replacing each space in the parse tree under the content of the S, with an NBSP. But note: the replacement should apply *not* to spaces in *all* text, but *only* to spaces in *printable* text. (This distinction may or may not be evident in the particular tree/event model implemented by the Pod parser.) For example, consider this unusual case:

```
S<L</Autoloaded Functions>>
```

This means that the space in the middle of the visible link text must not be broken across lines. In other words, it's the same as this:

```
L<"AutoloadedE<160>Functions"/Autoloaded Functions>
```

However, a misapplied space-to-NBSP replacement could (wrongly) produce something equivalent to this:

```
L<"AutoloadedE<160>Functions"/AutoloadedE<160>Functions>
```

...which is almost definitely not going to work as a hyperlink (assuming this formatter outputs a format supporting hypertext).

Formatters may choose to just not support the S format code, especially in cases where the output format simply has no NBSP character/code and no code for "don't break this stuff across lines".

- Besides the NBSP character discussed above, implementors are reminded of the existence of the other "special" character in Latin-1, the "soft hyphen" character, also known as "discretionary hyphen", i.e. E<173> = E<0xAD> = E<shy>). This character expresses an optional hyphenation point. That is, it normally renders as nothing, but may render as a "-" if a formatter breaks the word at that point. Pod formatters should, as appropriate, do one of the following: 1) render this with a code with the same meaning (e.g., "\-" in RTF), 2) pass it through in the expectation that the formatter understands this character as such, or 3) delete it.

For example:

```
sigE<shy>action
manuE<shy>script
JarkE<shy>ko HieE<shy>taE<shy>nieE<shy>mi
```

These signal to a formatter that if it is to hyphenate "sigaction" or "manuscript", then it should be done as "sig-[linebreak]action" or "manu-[linebreak]script" (and if it doesn't hyphenate it, then the E<shy> doesn't show up at all). And if it is to hyphenate "Jarkko" and/or "Hietaniemi", it can do so only at the points where there is a E<shy> code.

In practice, it is anticipated that this character will not be used often, but formatters should either support it, or delete it.

- If you think that you want to add a new command to Pod (like, say, a "=biblio" command), consider whether you could get the same effect with a for or begin/end sequence: "=for biblio ..." or "=begin biblio" ... "=end biblio". Pod processors that don't understand "=for biblio", etc, will simply ignore it, whereas they may complain loudly if they see "=biblio".
- Throughout this document, "Pod" has been the preferred spelling for the name of the documentation format. One may also use "POD" or "pod". For the documentation that is (typically) in the Pod format, you may use "pod", or "Pod", or "POD". Understanding these distinctions is useful; but obsessing over how to spell them, usually is not.

About L<...> Codes

As you can tell from a glance at [perlpod/perlpod](#), the L<... code is the most complex of the Pod formatting codes. The points below will hopefully clarify what it means and how processors should deal with it.

- In parsing an L<... code, Pod parsers must distinguish at least four attributes:

First:

The link-text. If there is none, this must be undef. (E.g., in "L<Perl Functions|perlfunc", the link-text is "Perl Functions". In "L<Time::HiRes" and even "L<|Time::HiRes", there is no link text. Note that link text may contain formatting.)

Second:

The possibly inferred link-text — i.e., if there was no real link text, then this is the text that we'll infer in its place. (E.g., for "L<Getopt::Std", the inferred link text is "Getopt::Std".)

Third:

The name or URL, or undef if none. (E.g., in "L<Perl Functions|perlfunc", the name — also sometimes called the page — is "perlfunc". In "L</CAVEATS", the name is undef.)

Fourth:

The section (AKA "item" in older perlpods), or undef if none. E.g., in [Getopt::Std/DESCRIPTION](#), "DESCRIPTION" is the section. (Note that this is not the same as a manpage section like the "5" in "man 5 crontab". "Section Foo" in the Pod sense means the part of the text that's introduced by the heading or item whose text is "Foo".)

Pod parsers may also note additional attributes including:

Fifth:

A flag for whether item 3 (if present) is a URL (like "http://lists.perl.org" is), in which case there should be no section attribute; a Pod name (like "perldoc" and "Getopt::Std" are); or possibly a man page name (like "crontab(5)" is).

Sixth:

The raw original L<... content, before text is split on "|", "/", etc, and before E<... codes are expanded.

(The above were numbered only for concise reference below. It is not a requirement that these be passed as an actual list or array.)

For example:

```
L<Foo::Bar>
=>  undef,                # link text
    "Foo::Bar",           # possibly inferred link text
    "Foo::Bar",           # name
    undef,                # section
    'pod',                # what sort of link
    "Foo::Bar"            # original content
```

```

L<Perlport's section on NL's|perlport/Newlines>
=>  "Perlport's section on NL's",      # link text
    "Perlport's section on NL's",      # possibly inferred link text
    "perlport",                        # name
    "Newlines",                        # section
    'pod',                             # what sort of link
    "Perlport's section on NL's|perlport/Newlines" # orig. content

L<perlport/Newlines>
=>  undef,                             # link text
    '"Newlines" in perlport',          # possibly inferred link text
    "perlport",                        # name
    "Newlines",                        # section
    'pod',                             # what sort of link
    "perlport/Newlines"                # original content

L<crontab(5)/"DESCRIPTION">
=>  undef,                             # link text
    '"DESCRIPTION" in crontab(5)',      # possibly inferred link text
    "crontab(5)",                      # name
    "DESCRIPTION",                     # section
    'man',                             # what sort of link
    'crontab(5)/"DESCRIPTION"'          # original content

L</Object Attributes>
=>  undef,                             # link text
    '"Object Attributes"',              # possibly inferred link text
    undef,                             # name
    "Object Attributes",                # section
    'pod',                             # what sort of link
    "/Object Attributes"                # original content

L<http://www.perl.org/>
=>  undef,                             # link text
    "http://www.perl.org/",             # possibly inferred link text
    "http://www.perl.org/",             # name
    undef,                             # section
    'url',                             # what sort of link
    "http://www.perl.org/"              # original content

```

Note that you can distinguish URL-links from anything else by the fact that they match `m/\A\w+:[^:\s]\S*\z/`. So `L<http://www.perl.com>` is a URL, but `L<HTTP::Response>` isn't.

- In case of `L<...>` codes with no "text|" part in them, older formatters have exhibited great variation in actually displaying the link or cross reference. For example, `L<crontab(5)>` would render as "the crontab(5) manpage", or "in the crontab(5) manpage" or just "crontab(5)".

Pod processors must now treat "text|"–less links as follows:

```

L<name>          =>  L<name|name>
L</section>      =>  L<"section"|/section>
L<name/section> =>  L<"section" in name|name/section>

```

- Note that section names might contain markup. I.e., if a section starts with:

```
=head2 About the C<-M> Operator
```

or with:

=item About the C<-M> Operator

then a link to it would look like this:

L<somedoc/About the C<-M> Operator>

Formatters may choose to ignore the markup for purposes of resolving the link and use only the renderable characters in the section name, as in:

```
<h1><a name="About_the_-M_Operator">About the <code>-M</code>
Operator</h1>
```

...

```
<a href="somedoc#About_the_-M_Operator">About the <code>-M</code>
Operator" in somedoc</a>
```

- Previous versions of perlpod distinguished L<name/"section"> links from L<name/item> links (and their targets). These have been merged syntactically and semantically in the current specification, and *section* can refer either to a "=head*n* Heading Content" command or to a "=item Item Content" command. This specification does not specify what behavior should be in the case of a given document having several things all seeming to produce the same *section* identifier (e.g., in HTML, several things all producing the same *anchorname* in <a name="*anchorname*"...</a elements). Where Pod processors can control this behavior, they should use the first such anchor. That is, L<Foo/Bar> refers to the *first* "Bar" section in Foo.

But for some processors/formats this cannot be easily controlled; as with the HTML example, the behavior of multiple ambiguous <a name="*anchorname*"...</a is most easily just left up to browsers to decide.

- Authors wanting to link to a particular (absolute) URL, must do so only with "L<scheme:..." codes (like L<http://www.perl.org>), and must not attempt "L<Some Site Name|scheme:..." codes. This restriction avoids many problems in parsing and rendering L<... codes.
- In a L<text | . . . > code, text may contain formatting codes for formatting or for E<... escapes, as in:

```
L<B<ummE<234>stuff> | . . . >
```

For L<...> codes without a "name|" part, only E<...> and Z<> codes may occur — no other formatting codes. That is, authors should not use "L<B<Foo::Bar>>".

Note, however, that formatting codes and Z<'s can occur in any and all parts of an L<... (i.e., in *name*, *section*, *text*, and *url*).

Authors must not nest L<... codes. For example, "L<The L<Foo::Bar man page" should be treated as an error.

- Note that Pod authors may use formatting codes inside the "text" part of "L<text|name" (and so on for L<text|/"sec").

In other words, this is valid:

```
Go read L<the docs on C<$.>|perlvar/"$. ">
```

Some output formats that do allow rendering "L<..." codes as hypertext, might not allow the link-text to be formatted; in that case, formatters will have to just ignore that formatting.

- At time of writing, L<name> values are of two types: either the name of a Pod page like L<Foo::Bar> (which might be a real Perl module or program in an @INC / PATH directory, or a .pod file in those places); or the name of a UNIX man page, like L<crontab(5)>. In theory, L<chmod> is ambiguous between a Pod page called "chmod", or the Unix man page "chmod" (in whatever man-section). However, the presence of a string in parens, as in "crontab(5)", is sufficient to signal that what is being discussed is not a Pod page, and so is presumably a UNIX man page. The

distinction is of no importance to many Pod processors, but some processors that render to hypertext formats may need to distinguish them in order to know how to render a given `L<foo>` code.

- Previous versions of perlpod allowed for a `L<section>` syntax (as in "`L<Object Attributes>`"), which was not easily distinguishable from `L<name>` syntax. This syntax is no longer in the specification, and has been replaced by the `L<"section">` syntax (where the quotes were formerly optional). Pod parsers should tolerate the `L<section>` syntax, for a while at least. The suggested heuristic for distinguishing `L<section>` from `L<name>` is that if it contains any whitespace, it's a *section*. Pod processors may warn about this being deprecated syntax.

About `=over...=back` Regions

`"=over"..."=back` regions are used for various kinds of list-like structures. (I use the term "region" here simply as a collective term for everything from the `"=over"` to the matching `"=back"`.)

- The non-zero numeric *indentlevel* in `"=over indentlevel" ... "=back"` is used for giving the formatter a clue as to how many "spaces" (ems, or roughly equivalent units) it should tab over, although many formatters will have to convert this to an absolute measurement that may not exactly match with the size of spaces (or M's) in the document's base font. Other formatters may have to completely ignore the number. The lack of any explicit *indentlevel* parameter is equivalent to an *indentlevel* value of 4. Pod processors may complain if *indentlevel* is present but is not a positive number matching `m/\A(\d*\.)?\d+\z/`.
- Authors of Pod formatters are reminded that `"=over" ... "=back"` may map to several different constructs in your output format. For example, in converting Pod to (X)HTML, it can map to any of `<ul...`, `<ol...`, `<dl...</dl>`, or `<blockquote...</blockquote>`. Similarly, `"=item"` can map to `` or `<dt>`.
- Each `"=over" ... "=back"` region should be one of the following:
 - An `"=over" ... "=back"` region containing only `"=item *"` commands, each followed by some number of ordinary/verbatim paragraphs, other nested `"=over" ... "=back"` regions, `"=for..."` paragraphs, and `"=begin"..."=end"` regions.
(Pod processors must tolerate a bare `"=item"` as if it were `"=item *"`.) Whether `"*"` is rendered as a literal asterisk, an "o", or as some kind of real bullet character, is left up to the Pod formatter, and may depend on the level of nesting.
 - An `"=over" ... "=back"` region containing only `m/\A=item\s+\d+\.?\s*\z/` paragraphs, each one (or each group of them) followed by some number of ordinary/verbatim paragraphs, other nested `"=over" ... "=back"` regions, `"=for..."` paragraphs, and/or `"=begin"..."=end"` codes. Note that the numbers must start at 1 in each section, and must proceed in order and without skipping numbers.
(Pod processors must tolerate lines like `"=item 1"` as if they were `"=item 1."`, with the period.)
 - An `"=over" ... "=back"` region containing only `"=item [text]"` commands, each one (or each group of them) followed by some number of ordinary/verbatim paragraphs, other nested `"=over" ... "=back"` regions, or `"=for..."` paragraphs, and `"=begin"..."=end"` regions.
The `"=item [text]"` paragraph should not match `m/\A=item\s+\d+\.?\s*\z/` or `m/\A=item\s+*\s*\z/`, nor should it match just `m/\A=item\s*\z/`.
 - An `"=over" ... "=back"` region containing no `"=item"` paragraphs at all, and containing only some number of ordinary/verbatim paragraphs, and possibly also some nested `"=over" ... "=back"` regions, `"=for..."` paragraphs, and `"=begin"..."=end"` regions. Such an itemless `"=over" ... "=back"` region in Pod is equivalent in meaning to a `<blockquote...</blockquote>` element in HTML.

Note that with all the above cases, you can determine which type of `"=over" ... "=back"` you have, by examining the first (non-`"=cut"`, non-`"=pod"`) Pod paragraph after the `"=over"` command.

- Pod formatters *must* tolerate arbitrarily large amounts of text in the "`=item text...`" paragraph. In practice, most such paragraphs are short, as in:

```
=item For cutting off our trade with all parts of the world
```

But they may be arbitrarily long:

```
=item For transporting us beyond seas to be tried for pretended offenses
```

```
=item He is at this time transporting large armies of foreign mercenaries to complete the works of death, desolation and tyranny, already begun with circumstances of cruelty and perfidy scarcely paralleled in the most barbarous ages, and totally unworthy the head of a civilized nation.
```

- Pod processors should tolerate "`=item *`" / "`=item number`" commands with no accompanying paragraph. The middle item is an example:

```
=over
```

```
=item 1
```

```
Pick up dry cleaning.
```

```
=item 2
```

```
=item 3
```

```
Stop by the store. Get Abba Zabas, Stoli, and cheap lawn chairs.
```

```
=back
```

- No "`=over`" ... "`=back`" region can contain headings. Processors may treat such a heading as an error.
- Note that an "`=over`" ... "`=back`" region should have some content. That is, authors should not have an empty region like this:

```
=over
```

```
=back
```

Pod processors seeing such a contentless "`=over`" ... "`=back`" region, may ignore it, or may report it as an error.

- Processors must tolerate an "`=over`" list that goes off the end of the document (i.e., which has no matching "`=back`"), but they may warn about such a list.
- Authors of Pod formatters should note that this construct:

```
=item Neque
```

```
=item Porro
```

```
=item Quisquam Est
```

```
Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.
```

```
=item Ut Enim
```

is semantically ambiguous, in a way that makes formatting decisions a bit difficult. On the one hand, it could be mention of an item "Neque", mention of another item "Porro", and mention of another item "Quisquam Est", with just the last one requiring the explanatory paragraph "Qui dolorem ipsum quia dolor..."; and then an item "Ut Enim". In that case, you'd want to format it like so:

Neque

Porro

Quisquam Est

Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.

Ut Enim

But it could equally well be a discussion of three (related or equivalent) items, "Neque", "Porro", and "Quisquam Est", followed by a paragraph explaining them all, and then a new item "Ut Enim". In that case, you'd probably want to format it like so:

Neque

Porro

Quisquam Est

Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.

Ut Enim

But (for the foreseeable future), Pod does not provide any way for Pod authors to distinguish which grouping is meant by the above "=item"-cluster structure. So formatters should format it like so:

Neque

Porro

Quisquam Est

Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.

Ut Enim

That is, there should be (at least roughly) equal spacing between items as between paragraphs (although that spacing may well be less than the full height of a line of text). This leaves it to the reader to use (con)textual cues to figure out whether the "Qui dolorem ipsum..." paragraph applies to the "Quisquam Est" item or to all three items "Neque", "Porro", and "Quisquam Est". While not an ideal situation, this is preferable to providing formatting cues that may be actually contrary to the author's intent.

About Data Paragraphs and "=begin/=end" Regions

Data paragraphs are typically used for inlining non-Pod data that is to be used (typically passed through) when rendering the document to a specific format:

```
=begin rtf
```

```
\par{\pard\qr\sa4500{\i Printed\~\chdate\~\chtime}\par}
```

```
=end rtf
```

The exact same effect could, incidentally, be achieved with a single "=for" paragraph:

```
=for rtf \par{\pard\qr\sa4500{\i Printed\~\chdate\~\chtime}\par}
```

(Although that is not formally a data paragraph, it has the same meaning as one, and Pod parsers may parse it as one.)

Another example of a data paragraph:

```
=begin html

I like <em>PIE</em>!

<hr>Especially pecan pie!

=end html
```

If these were ordinary paragraphs, the Pod parser would try to expand the "E" (in the first paragraph) as a formatting code, just like "E<lt>" or "E<eacute>". But since this is in a "`=begin identifier`"..."`=end identifier`" region *and* the identifier "html" doesn't begin have a ":" prefix, the contents of this region are stored as data paragraphs, instead of being processed as ordinary paragraphs (or if they began with a spaces and/or tabs, as verbatim paragraphs).

As a further example: At time of writing, no "biblio" identifier is supported, but suppose some processor were written to recognize it as a way of (say) denoting a bibliographic reference (necessarily containing formatting codes in ordinary paragraphs). The fact that "biblio" paragraphs were meant for ordinary processing would be indicated by prefacing each "biblio" identifier with a colon:

```
=begin :biblio

Wirth, Niklaus. 1976. I<Algorithms + Data Structures =
Programs.> Prentice-Hall, Englewood Cliffs, NJ.

=end :biblio
```

This would signal to the parser that paragraphs in this begin...end region are subject to normal handling as ordinary/verbatim paragraphs (while still tagged as meant only for processors that understand the "biblio" identifier). The same effect could be had with:

```
=for :biblio
Wirth, Niklaus. 1976. I<Algorithms + Data Structures =
Programs.> Prentice-Hall, Englewood Cliffs, NJ.
```

The ":" on these identifiers means simply "process this stuff normally, even though the result will be for some special target". I suggest that parser APIs report "biblio" as the target identifier, but also report that it had a ":" prefix. (And similarly, with the above "html", report "html" as the target identifier, and note the *lack* of a ":" prefix.)

Note that a "`=begin identifier`"..."`=end identifier`" region where *identifier* begins with a colon, *can* contain commands. For example:

```
=begin :biblio

Wirth's classic is available in several editions, including:

=for comment
  hm, check abebooks.com for how much used copies cost.

=over

=item

Wirth, Niklaus. 1975. I<Algorithmen und Datenstrukturen.>
Teubner, Stuttgart. [Yes, it's in German.]

=item

Wirth, Niklaus. 1976. I<Algorithms + Data Structures =
Programs.> Prentice-Hall, Englewood Cliffs, NJ.

=back

=end :biblio
```


Note, however, a "`=begin identifier`"..."`=end identifier`" region where *identifier* does *not* begin with a colon, should not directly contain "`=head1`" ... "`=head4`" commands, nor "`=over`", nor "`=back`", nor "`=item`". For example, this may be considered invalid:

```
=begin somedata
This is a data paragraph.
=head1 Don't do this!
This is a data paragraph too.
=end somedata
```

A Pod processor may signal that the above (specifically the "`=head1`" paragraph) is an error. Note, however, that the following should *not* be treated as an error:

```
=begin somedata
This is a data paragraph.
=cut
# Yup, this isn't Pod anymore.
sub excl { (rand() > .5) ? "hoo!" : "hah!" }
=pod
This is a data paragraph too.
=end somedata
```

And this too is valid:

```
=begin someformat
This is a data paragraph.
    And this is a data paragraph.
=begin someotherformat
This is a data paragraph too.
    And this is a data paragraph too.
=begin :yetanotherformat
=head2 This is a command paragraph!
This is an ordinary paragraph!
    And this is a verbatim paragraph!
=end :yetanotherformat
=end someotherformat
Another data paragraph!
=end someformat
```

The contents of the above "`=begin :yetanotherformat`" ... "`=end :yetanotherformat`" region *aren't* data paragraphs, because the immediately containing region's identifier ("`:yetanotherformat`") begins with a colon. In practice, most regions that contain data paragraphs will contain *only* data paragraphs; however, the above nesting is syntactically valid as Pod, even if it is rare. However, the handlers for some formats, like "`html`", will accept only data paragraphs, not nested regions; and they may complain if they see (targeted for them) nested regions, or commands, other than "`=end`", "`=pod`", and "`=cut`".

Also consider this valid structure:

```
=begin :biblio

Wirth's classic is available in several editions, including:

=over

=item

Wirth, Niklaus. 1975. I<Algorithmen und Datenstrukturen.>
Teubner, Stuttgart. [Yes, it's in German.]

=item

Wirth, Niklaus. 1976. I<Algorithms + Data Structures =
Programs.> Prentice-Hall, Englewood Cliffs, NJ.

=back

Buy buy buy!

=begin html

<img src='wirth_spokesmodeling_book.png'>

<hr>

=end html

Now now now!

=end :biblio
```

There, the "`=begin html`"..."`=end html`" region is nested inside the larger "`=begin :biblio`"..."`=end :biblio`" region. Note that the content of the "`=begin html`"..."`=end html`" region is data paragraph(s), because the immediately containing region's identifier ("`html`") *doesn't* begin with a colon.

Pod parsers, when processing a series of data paragraphs one after another (within a single region), should consider them to be one large data paragraph that happens to contain blank lines. So the content of the above "`=begin html`"..."`=end html`" *may* be stored as two data paragraphs (one consisting of "`<img src='wirth_spokesmodeling_book.png'\n`" and another consisting of "`<hr\n`"), but *should* be stored as a single data paragraph (consisting of "`<img src='wirth_spokesmodeling_book.png'\n\n<hr\n`").

Pod processors should tolerate empty "`=begin something`"..."`=end something`" regions, empty "`=begin :something`"..."`=end :something`" regions, and contentless "`=for something`" and "`=for :something`" paragraphs. I.e., these should be tolerated:

```
=for html

=begin html

=end html

=begin :biblio

=end :biblio
```

Incidentally, note that there's no easy way to express a data paragraph starting with something that looks like a command. Consider:

```
=begin stuff

=shazbot

=end stuff
```

There, "`=shazbot`" will be parsed as a Pod command "`shazbot`", not as a data paragraph "`=shazbot\n`".

However, you can express a data paragraph consisting of "`=shazbot\n`" using this code:

```
=for stuff =shazbot
```

The situation where this is necessary, is presumably quite rare.

Note that `=end` commands must match the currently open `=begin` command. That is, they must properly nest. For example, this is valid:

```
=begin outer
X
=begin inner
Y
=end inner
Z
=end outer
```

while this is invalid:

```
=begin outer
X
=begin inner
Y
=end outer
Z
=end inner
```

This latter is improper because when the "`=end outer`" command is seen, the currently open region has the formatname "inner", not "outer". (It just happens that "outer" is the format name of a higher-up region.) This is an error. Processors must by default report this as an error, and may halt processing the document containing that error. A corollary of this is that regions cannot "overlap" — i.e., the latter block above does not represent a region called "outer" which contains X and Y, overlapping a region called "inner" which contains Y and Z. But because it is invalid (as all apparently overlapping regions would be), it doesn't represent that, or anything at all.

Similarly, this is invalid:

```
=begin thing
=end hting
```

This is an error because the region is opened by "thing", and the "`=end`" tries to close "hting" [sic].

This is also invalid:

```
=begin thing
=end
```

This is invalid because every "`=end`" command must have a formatname parameter.

SEE ALSO

[perlpod](#), [PODs: Embedded Documentation in perlsyn](#), [podchecker](#)

AUTHOR

Sean M. Burke

NAME

perlport – Writing portable Perl

DESCRIPTION

Perl runs on numerous operating systems. While most of them share much in common, they also have their own unique features.

This document is meant to help you to find out what constitutes portable Perl code. That way once you make a decision to write portably, you know where the lines are drawn, and you can stay within them.

There is a tradeoff between taking full advantage of one particular type of computer and taking advantage of a full range of them. Naturally, as you broaden your range and become more diverse, the common factors drop, and you are left with an increasingly smaller area of common ground in which you can operate to accomplish a particular task. Thus, when you begin attacking a problem, it is important to consider under which part of the tradeoff curve you want to operate. Specifically, you must decide whether it is important that the task that you are coding have the full generality of being portable, or whether to just get the job done right now. This is the hardest choice to be made. The rest is easy, because Perl provides many choices, whichever way you want to approach your problem.

Looking at it another way, writing portable code is usually about willfully limiting your available choices. Naturally, it takes discipline and sacrifice to do that. The product of portability and convenience may be a constant. You have been warned.

Be aware of two important points:

Not all Perl programs have to be portable

There is no reason you should not use Perl as a language to glue Unix tools together, or to prototype a Macintosh application, or to manage the Windows registry. If it makes no sense to aim for portability for one reason or another in a given program, then don't bother.

Nearly all of Perl already *is* portable

Don't be fooled into thinking that it is hard to create portable Perl code. It isn't. Perl tries its level-best to bridge the gaps between what's available on different platforms, and all the means available to use those features. Thus almost all Perl code runs on any machine without modification. But there are some significant issues in writing portable code, and this document is entirely about those issues.

Here's the general rule: When you approach a task commonly done using a whole range of platforms, think about writing portable code. That way, you don't sacrifice much by way of the implementation choices you can avail yourself of, and at the same time you can give your users lots of platform choices. On the other hand, when you have to take advantage of some unique feature of a particular platform, as is often the case with systems programming (whether for Unix, Windows, Mac OS, VMS, etc.), consider writing platform-specific code.

When the code will run on only two or three operating systems, you may need to consider only the differences of those particular systems. The important thing is to decide where the code will run and to be deliberate in your decision.

The material below is separated into three main sections: main issues of portability ("*ISSUES*", platform-specific issues ("*PLATFORMS*", and built-in perl functions that behave differently on various ports ("*FUNCTION IMPLEMENTATIONS*").

This information should not be considered complete; it includes possibly transient information about idiosyncrasies of some of the ports, almost all of which are in a state of constant evolution. Thus, this material should be considered a perpetual work in progress (< <IMG SRC="yellow_sign.gif" ALT="Under Construction").

ISSUES

Newlines

In most operating systems, lines in files are terminated by newlines. Just what is used as a newline may vary from OS to OS. Unix traditionally uses `\012`, one type of DOSish I/O uses `\015\012`, and Mac OS uses `\015`.

Perl uses `\n` to represent the "logical" newline, where what is logical may depend on the platform in use. In MacPerl, `\n` always means `\015`. In DOSish perls, `\n` usually means `\012`, but when accessing a file in "text" mode, STDIO translates it to (or from) `\015\012`, depending on whether you're reading or writing. Unix does the same thing on ttys in canonical mode. `\015\012` is commonly referred to as CRLF.

A common cause of unportable programs is the misuse of `chop()` to trim newlines:

```
# XXX UNPORTABLE!
while(<FILE>) {
    chop;
    @array = split(/:/);
    #...
}
```

You can get away with this on Unix and Mac OS (they have a single character end-of-line), but the same program will break under DOSish perls because you're only `chop()`ing half the end-of-line. Instead, `chomp()` should be used to trim newlines. The `Dunce::Files` module can help audit your code for misuses of `chop()`.

When dealing with binary files (or text files in binary mode) be sure to explicitly set `$/` to the appropriate value for your file format before using `chomp()`.

Because of the "text" mode translation, DOSish perls have limitations in using `seek` and `tell` on a file accessed in "text" mode. Stick to `seek`-ing to locations you got from `tell` (and no others), and you are usually free to use `seek` and `tell` even in "text" mode. Using `seek` or `tell` or other file operations may be non-portable. If you use `binmode` on a file, however, you can usually `seek` and `tell` with arbitrary values in safety.

A common misconception in socket programming is that `\n` eq `\012` everywhere. When using protocols such as common Internet protocols, `\012` and `\015` are called for specifically, and the values of the logical `\n` and `\r` (carriage return) are not reliable.

```
print SOCKET "Hi there, client!\r\n";      # WRONG
print SOCKET "Hi there, client!\015\012";  # RIGHT
```

However, using `\015\012` (or `\cM\cJ`, or `\x0D\x0A`) can be tedious and unsightly, as well as confusing to those maintaining the code. As such, the `Socket` module supplies the Right Thing for those who want it.

```
use Socket qw(:DEFAULT :crlf);
print SOCKET "Hi there, client!$CRLF"      # RIGHT
```

When reading from a socket, remember that the default input record separator `$/` is `\n`, but robust socket code will recognize as either `\012` or `\015\012` as end of line:

```
while (<SOCKET>) {
    # ...
}
```

Because both CRLF and LF end in LF, the input record separator can be set to LF and any CR stripped later. Better to write:

```
use Socket qw(:DEFAULT :crlf);
local($/) = LF;          # not needed if $/ is already \012
```

```
while (<SOCKET>) {
    s/$CR?$LF/\n/;    # not sure if socket uses LF or CRLF, OK
    # s/\015?\012/\n/; # same thing
}
```

This example is preferred over the previous one—even for Unix platforms—because now any `\015`'s (`\cM`'s) are stripped out (and there was much rejoicing).

Similarly, functions that return text data—such as a function that fetches a web page—should sometimes translate newlines before returning the data, if they've not yet been translated to the local newline representation. A single line of code will often suffice:

```
$data =~ s/\015?\012/\n/g;
return $data;
```

Some of this may be confusing. Here's a handy reference to the ASCII CR and LF characters. You can print it out and stick it in your wallet.

LF	eq	\012	eq	\x0A	eq	\cJ	eq	chr(10)	eq	ASCII 10
CR	eq	\015	eq	\x0D	eq	\cM	eq	chr(13)	eq	ASCII 13

		Unix		DOS		Mac	

\n		LF		LF		CR	
\r		CR		CR		LF	
\n *		LF		CRLF		CR	
\r *		CR		CR		LF	

* text-mode STDIO							

The Unix column assumes that you are not accessing a serial line (like a tty) in canonical mode. If you are, then CR on input becomes `"\n"`, and `"\n"` on output becomes CRLF.

These are just the most common definitions of `\n` and `\r` in Perl. There may well be others. For example, on an EBCDIC implementation such as z/OS or OS/400 the above material is similar to "Unix" but the code numbers change:

LF	eq	\025	eq	\x15	eq		chr(21)	eq	CP-1047 21	
LF	eq	\045	eq	\x25	eq	\cU	eq	chr(37)	eq	CP-0037 37
CR	eq	\015	eq	\x0D	eq	\cM	eq	chr(13)	eq	CP-1047 13
CR	eq	\015	eq	\x0D	eq	\cM	eq	chr(13)	eq	CP-0037 13

		z/OS		OS/400	

\n		LF		LF	
\r		CR		CR	
\n *		LF		LF	
\r *		CR		CR	

* text-mode STDIO					

Numbers endianness and Width

Different CPUs store integers and floating point numbers in different orders (called *endianness*) and widths (32-bit and 64-bit being the most common today). This affects your programs when they attempt to transfer numbers in binary format from one CPU architecture to another, usually either "live" via network connection, or by storing the numbers to secondary storage such as a disk file or tape.

Conflicting storage orders make utter mess out of the numbers. If a little-endian host (Intel, VAX) stores 0x12345678 (305419896 in decimal), a big-endian host (Motorola, Sparc, PA) reads it as 0x78563412 (2018915346 in decimal). Alpha and MIPS can be either: Digital/Compaq used/uses them in little-endian

mode; SGI/Cray uses them in big-endian mode. To avoid this problem in network (socket) connections use the `pack` and `unpack` formats `n` and `N`, the "network" orders. These are guaranteed to be portable.

You can explore the endianness of your platform by unpacking a data structure packed in native format such as:

```
print unpack("h*", pack("s2", 1, 2)), "\n";
# '10002000' on e.g. Intel x86 or Alpha 21064 in little-endian mode
# '00100020' on e.g. Motorola 68040
```

If you need to distinguish between endian architectures you could use either of the variables set like so:

```
$is_big_endian    = unpack("h*", pack("s", 1)) =~ /01/;
$is_little_endian = unpack("h*", pack("s", 1)) =~ /^1/;
```

Differing widths can cause truncation even between platforms of equal endianness. The platform of shorter width loses the upper parts of the number. There is no good solution for this problem except to avoid transferring or storing raw binary numbers.

One can circumnavigate both these problems in two ways. Either transfer and store numbers always in text format, instead of raw binary, or else consider using modules like `Data::Dumper` (included in the standard distribution as of Perl 5.005) and `Storable` (included as of perl 5.8). Keeping all data as text significantly simplifies matters.

The v-strings are portable only up to `v2147483647` (`0x7FFFFFFF`), that's how far EBCDIC, or more precisely UTF-EBCDIC will go.

Files and Filesystems

Most platforms these days structure files in a hierarchical fashion. So, it is reasonably safe to assume that all platforms support the notion of a "path" to uniquely identify a file on the system. How that path is really written, though, differs considerably.

Although similar, file path specifications differ between Unix, Windows, Mac OS, OS/2, VMS, VOS, RISC OS, and probably others. Unix, for example, is one of the few OSes that has the elegant idea of a single root directory.

DOS, OS/2, VMS, VOS, and Windows can work similarly to Unix with `/` as path separator, or in their own idiosyncratic ways (such as having several root directories and various "unrooted" device files such as `NIL:` and `LPT:`).

Mac OS uses `:` as a path separator instead of `/`.

The filesystem may support neither hard links (`link`) nor symbolic links (`symlink`, `readlink`, `lstat`).

The filesystem may support neither access timestamp nor change timestamp (meaning that about the only portable timestamp is the modification timestamp), or one second granularity of any timestamps (e.g. the FAT filesystem limits the time granularity to two seconds).

The "inode change timestamp" (the `-C` filetest) may really be the "creation timestamp" (which it is not in UNIX).

VOS perl can emulate Unix filenames with `/` as path separator. The native pathname characters greater-than, less-than, number-sign, and percent-sign are always accepted.

RISC OS perl can emulate Unix filenames with `/` as path separator, or go native and use `.` for path separator and `:` to signal filesystems and disk names.

Don't assume UNIX filesystem access semantics: that read, write, and execute are all the permissions there are, and even if they exist, that their semantics (for example what do `r`, `w`, and `x` mean on a directory) are the UNIX ones. The various UNIX/POSIX compatibility layers usually try to make interfaces like `chmod()` work, but sometimes there simply is no good mapping.

If all this is intimidating, have no (well, maybe only a little) fear. There are modules that can help. The `File::Spec` modules provide methods to do the Right Thing on whatever platform happens to be running the program.

```
use File::Spec::Functions;
chdir(updir());          # go up one directory
$file = catfile(curdir(), 'temp', 'file.txt');
# on Unix and Win32, './temp/file.txt'
# on Mac OS, ':temp:file.txt'
# on VMS, '[.temp]file.txt'
```

`File::Spec` is available in the standard distribution as of version 5.004_05. `File::Spec::Functions` is only in `File::Spec` 0.7 and later, and some versions of perl come with version 0.6. If `File::Spec` is not updated to 0.7 or later, you must use the object-oriented interface from `File::Spec` (or upgrade `File::Spec`).

In general, production code should not have file paths hardcoded. Making them user-supplied or read from a configuration file is better, keeping in mind that file path syntax varies on different machines.

This is especially noticeable in scripts like Makefiles and test suites, which often assume `/` as a path separator for subdirectories.

Also of use is `File::Basename` from the standard distribution, which splits a pathname into pieces (base filename, full path to directory, and file suffix).

Even when on a single platform (if you can call Unix a single platform), remember not to count on the existence or the contents of particular system-specific files or directories, like `/etc/passwd`, `/etc/sendmail.conf`, `/etc/resolv.conf`, or even `/tmp/`. For example, `/etc/passwd` may exist but not contain the encrypted passwords, because the system is using some form of enhanced security. Or it may not contain all the accounts, because the system is using NIS. If code does need to rely on such a file, include a description of the file and its format in the code's documentation, then make it easy for the user to override the default location of the file.

Don't assume a text file will end with a newline. They should, but people forget.

Do not have two files or directories of the same name with different case, like *test.pl* and *Test.pl*, as many platforms have case-insensitive (or at least case-forgiving) filenames. Also, try not to have non-word characters (except for `.`) in the names, and keep them to the 8.3 convention, for maximum portability, onerous a burden though this may appear.

Likewise, when using the `AutoSplit` module, try to keep your functions to 8.3 naming and case-insensitive conventions; or, at the least, make it so the resulting files have a unique (case-insensitively) first 8 characters.

Whitespace in filenames is tolerated on most systems, but not all, and even on systems where it might be tolerated, some utilities might become confused by such whitespace.

Many systems (DOS, VMS) cannot have more than one `.` in their filenames.

Don't assume `<` won't be the first character of a filename. Always use `< <` explicitly to open a file for reading, or even better, use the three-arg version of `open`, unless you want the user to be able to specify a pipe `open`.

```
open(FILE, '<', $existing_file) or die $!;
```

If filenames might use strange characters, it is safest to open it with `sysopen` instead of `open`. `open` is magic and can translate characters like `<`, `< <`, and `|`, which may be the wrong thing to do. (Sometimes, though, it's the right thing.) Three-arg `open` can also help protect against this translation in cases where it is undesirable.

Don't use `:` as a part of a filename since many systems use that for their own semantics (Mac OS Classic for separating pathname components, many networking schemes and utilities for separating the nodename and the pathname, and so on). For the same reasons, avoid `@`, `;` and `|`.

Don't assume that in pathnames you can collapse two leading slashes `//` into one: some networking and clustering filesystems have special semantics for that. Let the operating system to sort it out.

The *portable filename characters* as defined by ANSI C are

```
a b c d e f g h i j k l m n o p q r t u v w x y z
A B C D E F G H I J K L M N O P Q R T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
. _ -
```

and the `"-"` shouldn't be the first character. If you want to be hypercorrect, stay case-insensitive and within the 8.3 naming convention (all the files and directories have to be unique within one directory if their names are lowercased and truncated to eight characters before the `.`, if any, and to three characters after the `.`, if any). (And do not use `.`s in directory names.)

System Interaction

Not all platforms provide a command line. These are usually platforms that rely primarily on a Graphical User Interface (GUI) for user interaction. A program requiring a command line interface might not work everywhere. This is probably for the user of the program to deal with, so don't stay up late worrying about it.

Some platforms can't delete or rename files held open by the system. Remember to `close` files when you are done with them. Don't `unlink` or `rename` an open file. Don't `tie` or `open` a file already tied or opened; `untie` or `close` it first.

Don't open the same file more than once at a time for writing, as some operating systems put mandatory locks on such files.

Don't assume that write/modify permission on a directory gives the right to add or delete files/directories in that directory. That is filesystem specific: in some filesystems you need write/modify permission also (or even just) in the file/directory itself. In some filesystems (AFS, DFS) the permission to add/delete directory entries is a completely separate permission.

Don't assume that a single `unlink` completely gets rid of the file: some filesystems (most notably the ones in VMS) have versioned filesystems, and `unlink()` removes only the most recent one (it doesn't remove all the versions because by default the native tools on those platforms remove just the most recent version, too). The portable idiom to remove all the versions of a file is

```
1 while unlink "file";
```

This will terminate if the file is undeleteable for some reason (protected, not there, and so on).

Don't count on a specific environment variable existing in `%ENV`. Don't count on `%ENV` entries being case-sensitive, or even case-preserving. Don't try to clear `%ENV` by saying `%ENV = ()`; or, if you really have to, make it conditional on `$^O ne 'VMS'` since in VMS the `%ENV` table is much more than a per-process key-value string table.

Don't count on signals or `%SIG` for anything.

Don't count on filename globbing. Use `opendir`, `readdir`, and `closedir` instead.

Don't count on per-program environment variables, or per-program current directories.

Don't count on specific values of `$!`.

Command names versus file pathnames

Don't assume that the name used to invoke a command or program with `system` or `exec` can also be used to test for the existence of the file that holds the executable code for that command or program. First, many systems have "internal" commands that are built-in to the shell or OS and while these commands can be invoked, there is no corresponding file. Second, some operating systems (e.g., Cygwin, DJGPP, OS/2, and VOS) have required suffixes for executable files; these suffixes are generally permitted on the command name but are not required. Thus, a command like "perl" might exist in a file named "perl", "perl.exe", or

"perl.pm", depending on the operating system. The variable "_exe" in the Config module holds the executable suffix, if any. Third, the VMS port carefully sets up \$^X and \$Config{perlpath} so that no further processing is required. This is just as well, because the matching regular expression used below would then have to deal with a possible trailing version number in the VMS file name.

To convert \$^X to a file pathname, taking account of the requirements of the various operating system possibilities, say:

```
use Config;
$thisperl = $^X;
if ($^O ne 'VMS')
    {$thisperl .= $Config{_exe} unless $thisperl =~ m/$Config{_exe}$/i;}
```

To convert \$Config{perlpath} to a file pathname, say:

```
use Config;
$thisperl = $Config{perlpath};
if ($^O ne 'VMS')
    {$thisperl .= $Config{_exe} unless $thisperl =~ m/$Config{_exe}$/i;}
```

Interprocess Communication (IPC)

In general, don't directly access the system in code meant to be portable. That means, no `system`, `exec`, `fork`, `pipe`, ```, `qx//`, `open` with a `|`, nor any of the other things that makes being a perl hacker worth being.

Commands that launch external processes are generally supported on most platforms (though many of them do not support any type of forking). The problem with using them arises from what you invoke them on. External tools are often named differently on different platforms, may not be available in the same location, might accept different arguments, can behave differently, and often present their results in a platform-dependent way. Thus, you should seldom depend on them to produce consistent results. (Then again, if you're calling `netstat -a`, you probably don't expect it to run on both Unix and CP/M.)

One especially common bit of Perl code is opening a pipe to **sendmail**:

```
open(MAIL, '|/usr/lib/sendmail -t')
    or die "cannot fork sendmail: $!";
```

This is fine for systems programming when sendmail is known to be available. But it is not fine for many non-Unix systems, and even some Unix systems that may not have sendmail installed. If a portable solution is needed, see the various distributions on CPAN that deal with it. `Mail::Mailer` and `Mail::Send` in the `MailTools` distribution are commonly used, and provide several mailing methods, including mail, sendmail, and direct SMTP (via `Net::SMTP`) if a mail transfer agent is not available. `Mail::Sendmail` is a standalone module that provides simple, platform-independent mailing.

The Unix System V IPC (`msg*`(), `sem*`(), `shm*`()) is not available even on all Unix platforms.

Do not use either the bare result of `pack("N", 10, 20, 30, 40)` or bare v-strings (such as `v10.20.30.40`) to represent IPv4 addresses: both forms just pack the four bytes into network order. That this would be equal to the C language `in_addr` struct (which is what the socket code internally uses) is not guaranteed. To be portable use the routines of the `Socket` extension, such as `inet_aton()`, `inet_ntoa()`, and `sockaddr_in()`.

The rule of thumb for portable code is: Do it all in portable Perl, or use a module (that may internally implement it with platform-specific code, but expose a common interface).

External Subroutines (XS)

XS code can usually be made to work with any platform, but dependent libraries, header files, etc., might not be readily available or portable, or the XS code itself might be platform-specific, just as Perl code might be. If the libraries and headers are portable, then it is normally reasonable to make sure the XS code is portable, too.

A different type of portability issue arises when writing XS code: availability of a C compiler on the

end-user's system. C brings with it its own portability issues, and writing XS code will expose you to some of those. Writing purely in Perl is an easier way to achieve portability.

Standard Modules

In general, the standard modules work across platforms. Notable exceptions are the CPAN module (which currently makes connections to external programs that may not be available), platform-specific modules (like ExtUtils::MM_VMS), and DBM modules.

There is no one DBM module available on all platforms. SDBM_File and the others are generally available on all Unix and DOSish ports, but not in MacPerl, where only NDBM_File and DB_File are available.

The good news is that at least some DBM module should be available, and AnyDBM_File will use whichever module it can find. Of course, then the code needs to be fairly strict, dropping to the greatest common factor (e.g., not exceeding 1K for each record), so that it will work with any DBM module. See [AnyDBM_File](#) for more details.

Time and Date

The system's notion of time of day and calendar date is controlled in widely different ways. Don't assume the timezone is stored in `$ENV{TZ}`, and even if it is, don't assume that you can control the timezone through that variable.

Don't assume that the epoch starts at 00:00:00, January 1, 1970, because that is OS- and implementation-specific. It is better to store a date in an unambiguous representation. The ISO-8601 standard defines "YYYY-MM-DD" as the date format. A text representation (like "1987-12-18") can be easily converted into an OS-specific value using a module like Date::Parse. An array of values, such as those returned by `localtime`, can be converted to an OS-specific representation using `Time::Local`.

When calculating specific times, such as for tests in time or date modules, it may be appropriate to calculate an offset for the epoch.

```
require Time::Local;
$offset = Time::Local::timegm(0, 0, 0, 1, 0, 70);
```

The value for `$offset` in Unix will be , but in Mac OS will be some large number. `$offset` can then be added to a Unix time value to get what should be the proper value on any system.

Character sets and character encoding

Assume very little about character sets.

Assume nothing about numerical values (`ord`, `chr`) of characters. Do not use explicit code point ranges (like `\xHH-\xHH`); use for example symbolic character classes like `[:print:]`.

Do not assume that the alphabetic characters are encoded contiguously (in the numeric sense). There may be gaps.

Do not assume anything about the ordering of the characters. The lowercase letters may come before or after the uppercase letters; the lowercase and uppercase may be interlaced so that both 'a' and 'A' come before 'b'; the accented and other international characters may be interlaced so that ä comes before 'b'.

Internationalisation

If you may assume POSIX (a rather large assumption), you may read more about the POSIX locale system from [perllocale](#). The locale system at least attempts to make things a little bit more portable, or at least more convenient and native-friendly for non-English users. The system affects character sets and encoding, and date and time formatting—amongst other things.

System Resources

If your code is destined for systems with severely constrained (or missing!) virtual memory systems then you want to be *especially* mindful of avoiding wasteful constructs such as:

```
# NOTE: this is no longer "bad" in perl5.005
for (0..10000000) {}                                # bad
```

```
for (my $x = 0; $x <= 10000000; ++$x) {}    # good
@lines = <VERY_LARGE_FILE>;                # bad
while (<FILE>) {$file .= $_}                # sometimes bad
$file = join('', <FILE>);                   # better
```

The last two constructs may appear unintuitive to most people. The first repeatedly grows a string, whereas the second allocates a large chunk of memory in one go. On some systems, the second is more efficient than the first.

Security

Most multi-user platforms provide basic levels of security, usually implemented at the filesystem level. Some, however, do not—unfortunately. Thus the notion of user id, or "home" directory, or even the state of being logged-in, may be unrecognizable on many platforms. If you write programs that are security-conscious, it is usually best to know what type of system you will be running under so that you can write code explicitly for that platform (or class of platforms).

Don't assume the UNIX filesystem access semantics: the operating system or the filesystem may be using some ACL systems, which are richer languages than the usual `rxw`. Even if the `rxw` exist, their semantics might be different.

(From security viewpoint testing for permissions before attempting to do something is silly anyway: if one tries this, there is potential for race conditions—someone or something might change the permissions between the permissions check and the actual operation. Just try the operation.)

Don't assume the UNIX user and group semantics: especially, don't expect the `< $<` and `< $` (or the `$ (` and `)`) to work for switching identities (or memberships).

Don't assume `set-uid` and `set-gid` semantics. (And even if you do, think twice: `set-uid` and `set-gid` are a known can of security worms.)

Style

For those times when it is necessary to have platform-specific code, consider keeping the platform-specific code in one place, making porting to other platforms easier. Use the `Config` module and the special variable `^O` to differentiate platforms, as described in *"PLATFORMS"*.

Be careful in the tests you supply with your module or programs. Module code may be fully portable, but its tests might not be. This often happens when tests spawn off other processes or call external programs to aid in the testing, or when (as noted above) the tests assume certain things about the filesystem and paths. Be careful not to depend on a specific output style for errors, such as when checking `!` after a system call. Some platforms expect a certain output format, and perl on those platforms may have been adjusted accordingly. Most specifically, don't anchor a regex when testing an error value.

CPAN Testers

Modules uploaded to CPAN are tested by a variety of volunteers on different platforms. These CPAN testers are notified by mail of each new upload, and reply to the list with `PASS`, `FAIL`, `NA` (not applicable to this platform), or `UNKNOWN` (unknown), along with any relevant notations.

The purpose of the testing is twofold: one, to help developers fix any problems in their code that crop up because of lack of testing on other platforms; two, to provide users with information about whether a given module works on a given platform.

Mailing list: cpan-testers@perl.org

Testing results: <http://testers.cpan.org/>

PLATFORMS

As of version 5.002, Perl is built with a `^O` variable that indicates the operating system it was built on. This was implemented to help speed up code that would otherwise have to use `Config` and use the value of `$Config{osname}`. Of course, to get more detailed information about the system, looking into `%Config` is certainly recommended.

%Config cannot always be trusted, however, because it was built at compile time. If perl was built in one place, then transferred elsewhere, some values may be wrong. The values may even have been edited after the fact.

Unix

Perl works on a bewildering variety of Unix and Unix-like platforms (see e.g. most of the files in the *hints/* directory in the source code kit). On most of these systems, the value of `$^O` (hence `$Config{'osname'}`, too) is determined either by lowercasing and stripping punctuation from the first field of the string returned by typing `uname -a` (or a similar command) at the shell prompt or by testing the file system for the presence of uniquely named files such as a kernel or header file. Here, for example, are a few of the more popular Unix flavors:

uname	\$^O	\$Config{'archname'}
-----	-----	-----
AIX	aix	aix
BSD/OS	bsdos	i386-bsdos
Darwin	darwin	darwin
dgux	dgux	AViiON-dgux
DYNIX/ptx	dynixptx	i386-dynixptx
FreeBSD	freebsd	freebsd-i386
Linux	linux	arm-linux
Linux	linux	i386-linux
Linux	linux	i586-linux
Linux	linux	ppc-linux
HP-UX	hpux	PA-RISC1.1
IRIX	irix	irix
Mac OS X	darwin	darwin
MachTen PPC	machten	powerpc-machten
NeXT 3	next	next-fat
NeXT 4	next	OPENSTEP-Mach
openbsd	openbsd	i386-openbsd
OSF1	dec_osf	alpha-dec_osf
reliantunix-n	svr4	RM400-svr4
SCO_SV	sco_sv	i386-sco_sv
SINIX-N	svr4	RM400-svr4
sn4609	unicos	CRAY_C90-unicos
sn6521	unicosmk	t3e-unicosmk
sn9617	unicos	CRAY_J90-unicos
SunOS	solaris	sun4-solaris
SunOS	solaris	i86pc-solaris
SunOS4	sunos	sun4-sunos

Because the value of `$Config{archname}` may depend on the hardware architecture, it can vary more than the value of `$^O`.

DOS and Derivatives

Perl has long been ported to Intel-style microcomputers running under systems like PC-DOS, MS-DOS, OS/2, and most Windows platforms you can bring yourself to mention (except for Windows CE, if you count that). Users familiar with *COMMAND.COM* or *CMD.EXE* style shells should be aware that each of these file specifications may have subtle differences:

```
$filespec0 = "c:/foo/bar/file.txt";
$filespec1 = "c:\\foo\\bar\\file.txt";
$filespec2 = 'c:\foo\bar\file.txt';
$filespec3 = 'c:\\foo\\bar\\file.txt';
```

System calls accept either / or \ as the path separator. However, many command-line utilities of DOS

vintage treat / as the option prefix, so may get confused by filenames containing /. Aside from calling any external programs, / will work just fine, and probably better, as it is more consistent with popular usage, and avoids the problem of remembering what to backwhack and what not to.

The DOS FAT filesystem can accommodate only "8.3" style filenames. Under the "case-insensitive, but case-preserving" HPFS (OS/2) and NTFS (NT) filesystems you may have to be careful about case returned with functions like `readdir` or used with functions like `open` or `opendir`.

DOS also treats several filenames as special, such as AUX, PRN, NUL, CON, COM1, LPT1, LPT2, etc. Unfortunately, sometimes these filenames won't even work if you include an explicit directory prefix. It is best to avoid such filenames, if you want your code to be portable to DOS and its derivatives. It's hard to know what these all are, unfortunately.

Users of these operating systems may also wish to make use of scripts such as *pl2bat.bat* or *pl2cmd* to put wrappers around your scripts.

Newline (`\n`) is translated as `\015\012` by STDIO when reading from and writing to files (see *"Newlines"*). `binmode(FILEHANDLE)` will keep `\n` translated as `\012` for that filehandle. Since it is a no-op on other systems, `binmode` should be used for cross-platform code that deals with binary data. That's assuming you realize in advance that your data is in binary. General-purpose programs should often assume nothing about their data.

The `^O` variable and the `$Config{archname}` values for various DOSish perls are as follows:

OS	<code>^O</code>	<code>\$Config{archname}</code>	ID	Version
MS-DOS	<code>dos</code>	?		
PC-DOS	<code>dos</code>	?		
OS/2	<code>os2</code>	?		
Windows 3.1	?	?	0	3 01
Windows 95	<code>MSWin32</code>	<code>MSWin32-x86</code>	1	4 00
Windows 98	<code>MSWin32</code>	<code>MSWin32-x86</code>	1	4 10
Windows ME	<code>MSWin32</code>	<code>MSWin32-x86</code>	1	?
Windows NT	<code>MSWin32</code>	<code>MSWin32-x86</code>	2	4 xx
Windows NT	<code>MSWin32</code>	<code>MSWin32-ALPHA</code>	2	4 xx
Windows NT	<code>MSWin32</code>	<code>MSWin32-ppc</code>	2	4 xx
Windows 2000	<code>MSWin32</code>	<code>MSWin32-x86</code>	2	5 xx
Windows XP	<code>MSWin32</code>	<code>MSWin32-x86</code>	2	?
Windows CE	<code>MSWin32</code>	?	3	
Cygwin	<code>cygwin</code>	?		

The various MSWin32 Perl's can distinguish the OS they are running on via the value of the fifth element of the list returned from `Win32::GetOSVersion()`. For example:

```
if ($^O eq 'MSWin32') {
    my @os_version_info = Win32::GetOSVersion();
    print +('3.1','95','NT')[$os_version_info[4]], "\n";
}
```

There are also `Win32::IsWinNT()` and `Win32::IsWin95()`, try `perldoc Win32`, and as of `libwin32 0.19` (not part of the core Perl distribution) `Win32::GetOSName()`. The very portable `POSIX::uname()` will work too:

```
c:\> perl -MPOSIX -we "print join '|', uname"
Windows NT|moonru|5.0|Build 2195 (Service Pack 2)|x86
```

Also see:

- The djgpp environment for DOS, <http://www.delorie.com/djgpp/> and [perldos](#).
- The EMX environment for DOS, OS/2, etc. emx@iaehv.nl, <http://www.leo.org/pub/comp/os/os2/leo/gnu/emx+gcc/index.html> or <ftp://hobbes.nmsu.edu/pub/os2/dev/emx/> Also [perlos2](#).
- Build instructions for Win32 in [perlwin32](#), or under the Cygnus environment in [perlcygwin](#).
- The Win32::* modules in [Win32](#).
- The ActiveState Pages, <http://www.activestate.com/>
- The Cygwin environment for Win32; **README.cygwin** (installed as [perlcygwin](#)), <http://www.cygwin.com/>
- The U/WIN environment for Win32, <http://www.research.att.com/sw/tools/uwin/>
- Build instructions for OS/2, [perlos2](#)

Mac OS

Any module requiring XS compilation is right out for most people, because MacPerl is built using non-free (and non-cheap!) compilers. Some XS modules that can work with MacPerl are built and distributed in binary form on CPAN.

Directories are specified as:

volume:folder:file	for absolute pathnames
volume:folder:	for absolute pathnames
:folder:file	for relative pathnames
:folder:	for relative pathnames
:file	for relative pathnames
file	for relative pathnames

Files are stored in the directory in alphabetical order. Filenames are limited to 31 characters, and may include any character except for null and :, which is reserved as the path separator.

Instead of flock, see FSpSetFlock and FSpRstFlock in the Mac::Files module, or `chmod(0444, ...)` and `chmod(0666, ...)`.

In the MacPerl application, you can't run a program from the command line; programs that expect @ARGV to be populated can be edited with something like the following, which brings up a dialog box asking for the command line arguments.

```
if (!@ARGV) {
    @ARGV = split /\s+/, MacPerl::Ask('Arguments?');
}
```

A MacPerl script saved as a "droplet" will populate @ARGV with the full pathnames of the files dropped onto the script.

Mac users can run programs under a type of command line interface under MPW (Macintosh Programmer's Workshop, a free development environment from Apple). MacPerl was first introduced as an MPW tool, and MPW can be used like a shell:

```
perl myscript.plx some arguments
```

ToolServer is another app from Apple that provides access to MPW tools from MPW and the MacPerl app, which allows MacPerl programs to use `system`, backticks, and piped open.

"Mac OS" is the proper name for the operating system, but the value in `$^O` is "MacOS". To determine architecture, version, or whether the application or MPW tool version is running, check:

```
$is_app    = $MacPerl::Version =~ /App/;
$is_tool   = $MacPerl::Version =~ /MPW/;
```



```
($version) = $MacPerl::Version =~ /^(\S+)/;
$sis_ppc    = $MacPerl::Architecture eq 'MacPPC';
$sis_68k    = $MacPerl::Architecture eq 'Mac68K';
```

Mac OS X, based on NeXT's OpenStep OS, runs MacPerl natively, under the "Classic" environment. There is no "Carbon" version of MacPerl to run under the primary Mac OS X environment. Mac OS X and its Open Source version, Darwin, both run Unix perl natively.

Also see:

- MacPerl Development, <http://dev.macperl.org/> .
- The MacPerl Pages, <http://www.macperl.com/> .
- The MacPerl mailing lists, <http://lists.perl.org/> .

VMS

Perl on VMS is discussed in [perlvms](#) in the perl distribution. Perl on VMS can accept either VMS- or Unix-style file specifications as in either of the following:

```
$ perl -ne "print if /perl_setup/i" SYS$LOGIN:LOGIN.COM
$ perl -ne "print if /perl_setup/i" /sys$login/login.com
```

but not a mixture of both as in:

```
$ perl -ne "print if /perl_setup/i" sys$login:/login.com
Can't open sys$login:/login.com: file specification syntax error
```

Interacting with Perl from the Digital Command Language (DCL) shell often requires a different set of quotation marks than Unix shells do. For example:

```
$ perl -e "print \"Hello, world.\n\""
Hello, world.
```

There are several ways to wrap your perl scripts in DCL **.COM** files, if you are so inclined. For example:

```
$ write sys$output "Hello from DCL!"
$ if p1 .eqs. ""
$ then perl -x 'f$environment("PROCEDURE")
$ else perl -x - 'p1 'p2 'p3 'p4 'p5 'p6 'p7 'p8
$ deck/dollars="__END__"
#!/usr/bin/perl

print "Hello from Perl!\n";

__END__
$ endif
```

Do take care with \$ ASSIGN/nolog/user SYS\$COMMAND: SYS\$INPUT if your perl-in-DCL script expects to do things like < \$read = <STDIN; .

Filenames are in the format "name.extension;version". The maximum length for filenames is 39 characters, and the maximum length for extensions is also 39 characters. Version is a number from 1 to 32767. Valid characters are / [A-Z0-9\$_] / .

VMS's RMS filesystem is case-insensitive and does not preserve case. `readdir` returns lowercased filenames, but specifying a file for opening remains case-insensitive. Files without extensions have a trailing period on them, so doing a `readdir` with a file named **A.5** will return **a**. (though that file could be opened with `open (FH, 'A')`).

RMS had an eight level limit on directory depths from any rooted logical (allowing 16 levels overall) prior to VMS 7.2. Hence `PERL_ROOT:[LIB.2.3.4.5.6.7.8]` is a valid directory specification but `PERL_ROOT:[LIB.2.3.4.5.6.7.8.9]` is not. **Makefile.PL** authors might have to take this into account, but at least they can refer to the former as `/PERL_ROOT/lib/2/3/4/5/6/7/8/`.

The VMS::Filespec module, which gets installed as part of the build process on VMS, is a pure Perl module that can easily be installed on non-VMS platforms and can be helpful for conversions to and from RMS native formats.

What `\n` represents depends on the type of file opened. It usually represents `\012` but it could also be `\015`, `\012`, `\015\012`, `\000`, `\040`, or nothing depending on the file organization and record format. The VMS::Stdio module provides access to the special `fopen()` requirements of files with unusual attributes on VMS.

TCP/IP stacks are optional on VMS, so socket routines might not be implemented. UDP sockets may not be supported.

The value of `$^O` on OpenVMS is "VMS". To determine the architecture that you are running on without resorting to loading all of `%Config` you can examine the content of the `@INC` array like so:

```
if (grep(/VMS_AXP/, @INC)) {
    print "I'm on Alpha!\n";
} elsif (grep(/VMS_VAX/, @INC)) {
    print "I'm on VAX!\n";
} else {
    print "I'm not so sure about where $^O is...\n";
}
```

On VMS, perl determines the UTC offset from the `SYS$TIMEZONE_DIFFERENTIAL` logical name. Although the VMS epoch began at 17-NOV-1858 00:00:00.00, calls to `localtime` are adjusted to count offsets from 01-JAN-1970 00:00:00.00, just like Unix.

Also see:

- **README.vms** (installed as [README_vms](#)), [perlvm](#)s
- vmsperl list, majordomo@perl.org
(Put the words `subscribe vmsperl` in message body.)
- vmsperl on the web, <http://www.sidhe.org/vmsperl/index.html>

VOS

Perl on VOS is discussed in **README.vos** in the perl distribution (installed as [perlvos](#)). Perl on VOS can accept either VOS- or Unix-style file specifications as in either of the following:

```
C<< $ perl -ne "print if /perl_setup/i" >system>notices >>
C<< $ perl -ne "print if /perl_setup/i" /system/notices >>
```

or even a mixture of both as in:

```
C<< $ perl -ne "print if /perl_setup/i" >system/notices >>
```

Even though VOS allows the slash character to appear in object names, because the VOS port of Perl interprets it as a pathname delimiting character, VOS files, directories, or links whose names contain a slash character cannot be processed. Such files must be renamed before they can be processed by Perl. Note that VOS limits file names to 32 or fewer characters.

Perl on VOS can be built using two different compilers and two different versions of the POSIX runtime. The recommended method for building full Perl is with the GNU C compiler and the generally-available version of VOS POSIX support. See **README.vos** (installed as [perlvos](#)) for restrictions that apply when Perl is built using the VOS Standard C compiler or the alpha version of VOS POSIX support.

The value of `$^O` on VOS is "VOS". To determine the architecture that you are running on without resorting to loading all of `%Config` you can examine the content of the `@INC` array like so:

```
if ($^O =~ /VOS/) {
```

```

        print "I'm on a Stratus box!\n";
    } else {
        print "I'm not on a Stratus box!\n";
        die;
    }

    if (grep(/860/, @INC)) {
        print "This box is a Stratus XA/R!\n";
    } elsif (grep(/7100/, @INC)) {
        print "This box is a Stratus HP 7100 or 8xxx!\n";
    } elsif (grep(/8000/, @INC)) {
        print "This box is a Stratus HP 8xxx!\n";
    } else {
        print "This box is a Stratus 68K!\n";
    }

```

Also see:

- **README.vos** (installed as [perlvos](#))
- The VOS mailing list.

There is no specific mailing list for Perl on VOS. You can post comments to the comp.sys.stratus newsgroup, or subscribe to the general Stratus mailing list. Send a letter with "subscribe Info-Stratus" in the message body to majordomo@list.stratagy.com.

- VOS Perl on the web at <http://ftp.stratus.com/pub/vos/posix/posix.html>

EBCDIC Platforms

Recent versions of Perl have been ported to platforms such as OS/400 on AS/400 minicomputers as well as OS/390, VM/ESA, and BS2000 for S/390 Mainframes. Such computers use EBCDIC character sets internally (usually Character Code Set ID 0037 for OS/400 and either 1047 or POSIX-BC for S/390 systems). On the mainframe perl currently works under the "Unix system services for OS/390" (formerly known as OpenEdition), VM/ESA OpenEdition, or the BS200 POSIX-BC system (BS2000 is supported in perl 5.6 and greater). See [perlos390](#) for details.

As of R2.5 of USS for OS/390 and Version 2.3 of VM/ESA these Unix sub-systems do not support the #! shebang trick for script invocation. Hence, on OS/390 and VM/ESA perl scripts can be executed with a header similar to the following simple script:

```

: # use perl
  eval 'exec /usr/local/bin/perl -S $0 ${1+"$@"}'
  if 0;
#!/usr/local/bin/perl      # just a comment really
print "Hello from perl!\n";

```

OS/390 will support the #! shebang trick in release 2.8 and beyond. Calls to `system` and backticks can use POSIX shell syntax on all S/390 systems.

On the AS/400, if PERL5 is in your library list, you may need to wrap your perl scripts in a CL procedure to invoke them like so:

```

BEGIN
  CALL PGM(PERL5/PERL) PARM('/QOpenSys/hello.pl')
ENDPGM

```

This will invoke the perl script **hello.pl** in the root of the QOpenSys file system. On the AS/400 calls to `system` or backticks must use CL syntax.

On these platforms, bear in mind that the EBCDIC character set may have an effect on what happens with some perl functions (such as `chr`, `pack`, `print`, `printf`, `ord`, `sort`, `sprintf`, `unpack`), as well as bit-fiddling with ASCII constants using operators like `^`, `&` and `|`, not to mention dealing with socket interfaces to ASCII computers (see ["Newlines"](#)).

Fortunately, most web servers for the mainframe will correctly translate the `\n` in the following statement to its ASCII equivalent (`\r` is the same under both Unix and OS/390 & VM/ESA):

```
print "Content-type: text/html\r\n\r\n";
```

The values of `$^O` on some of these platforms includes:

uname	<code>\$^O</code>	<code>\$Config{'archname'}</code>
OS/390	os390	os390
OS400	os400	os400
POSIX-BC	posix-bc	BS2000-posix-bc
VM/ESA	vmesa	vmesa

Some simple tricks for determining if you are running on an EBCDIC platform could include any of the following (perhaps all):

```
if ("\t" eq "\05") { print "EBCDIC may be spoken here!\n"; }
if (ord('A') == 193) { print "EBCDIC may be spoken here!\n"; }
if (chr(169) eq 'z') { print "EBCDIC may be spoken here!\n"; }
```

One thing you may not want to rely on is the EBCDIC encoding of punctuation characters since these may differ from code page to code page (and once your module or script is rumoured to work with EBCDIC, folks will want it to work with all EBCDIC character sets).

Also see:

- [*](#)
[perlos390](#), [README.os390](#), [perlbs2000](#), [README.vmesa](#), [perlebcdic](#).
- The `perl-mvs@perl.org` list is for discussion of porting issues as well as general usage issues for all EBCDIC Perls. Send a message body of "subscribe perl-mvs" to `majordomo@perl.org`.
- AS/400 Perl information at <http://as400.rochester.ibm.com/> as well as on CPAN in the [ports/](#) directory.

Acorn RISC OS

Because Acorns use ASCII with newlines (`\n`) in text files as `\012` like Unix, and because Unix filename emulation is turned on by default, most simple scripts will probably work "out of the box". The native filesystem is modular, and individual filesystems are free to be case-sensitive or insensitive, and are usually case-preserving. Some native filesystems have name length limits, which file and directory names are silently truncated to fit. Scripts should be aware that the standard filesystem currently has a name length limit of **10** characters, with up to 77 items in a directory, but other filesystems may not impose such limitations.

Native filenames are of the form

```
Filesystem#Special_Field::DiskName.$Directory.Directory.File
```

where

```
Special_Field is not usually present, but may contain . and $ .
Filesystem =~ m/[A-Za-z0-9_]/
DiskName    =~ m/[A-Za-z0-9_\/]/
$ represents the root directory
. is the path separator
```

```
@ is the current directory (per filesystem but machine global)
^ is the parent directory
Directory and File =~ m|[^\\0- "\.\\$%\\&:\\@\\^\\|\\177]+|
```

The default filename translation is roughly `tr|/.|./|;`

Note that "ADFS::HardDisk.\$File" ne 'ADFS::HardDisk.\$File' and that the second stage of \$ interpolation in regular expressions will fall foul of the \$. if scripts are not careful.

Logical paths specified by system variables containing comma-separated search lists are also allowed; hence System:Modules is a valid filename, and the filesystem will prefix Modules with each section of System\$Path until a name is made that points to an object on disk. Writing to a new file System:Modules would be allowed only if System\$Path contains a single item list. The filesystem will also expand system variables in filenames if enclosed in angle brackets, so <

<System\$Dir.Modules would look for the file \$ENV{'System\$Dir'} . 'Modules'. The obvious implication of this is that **fully qualified filenames can start with < <** and should be protected when open is used for input.

Because . was in use as a directory separator and filenames could not be assumed to be unique after 10 characters, Acorn implemented the C compiler to strip the trailing .c .h .s and .o suffix from filenames specified in source code and store the respective files in subdirectories named after the suffix. Hence files are translated:

foo.h	h.foo	
C:foo.h	C:h.foo	(logical path variable)
sys/os.h	sys.h.os	(C compiler groks Unix-speak)
10charname.c	c.10charname	
10charname.o	o.10charname	
11charname_.c	c.11charname	(assuming filesystem truncates at 10)

The Unix emulation library's translation of filenames to native assumes that this sort of translation is required, and it allows a user-defined list of known suffixes that it will transpose in this fashion. This may seem transparent, but consider that with these rules foo/bar/baz.h and foo/bar/h/baz both map to foo.bar.h.baz, and that readdir and glob cannot and do not attempt to emulate the reverse mapping. Other .'s in filenames are translated to /.

As implied above, the environment accessed through %ENV is global, and the convention is that program specific environment variables are of the form Program\$Name. Each filesystem maintains a current directory, and the current filesystem's current directory is the **global** current directory. Consequently, sociable programs don't change the current directory but rely on full pathnames, and programs (and Makefiles) cannot assume that they can spawn a child process which can change the current directory without affecting its parent (and everyone else for that matter).

Because native operating system filehandles are global and are currently allocated down from 255, with 0 being a reserved value, the Unix emulation library emulates Unix filehandles. Consequently, you can't rely on passing STDIN, STDOUT, or STDERR to your children.

The desire of users to express filenames of the form < <Foo\$Dir.Bar on the command line unquoted causes problems, too: `` command output capture has to perform a guessing game. It assumes that a string < <[^<]+\\${[^<]} is a reference to an environment variable, whereas anything else involving < < or < is redirection, and generally manages to be 99% right. Of course, the problem remains that scripts cannot rely on any Unix tools being available, or that any tools found have Unix-like command line arguments.

Extensions and XS are, in theory, buildable by anyone using free tools. In practice, many don't, as users of the Acorn platform are used to binary distributions. MakeMaker does run, but no available make currently copes with MakeMaker's makefiles; even if and when this should be fixed, the lack of a Unix-like shell will cause problems with makefile rules, especially lines of the form `cd sdbm && make all`, and anything using quoting.

"RISC OS" is the proper name for the operating system, but the value in `$^O` is "riscos" (because we don't like shouting).

Other perls

Perl has been ported to many platforms that do not fit into any of the categories listed above. Some, such as AmigaOS, Atari MiNT, BeOS, HP MPE/iX, QNX, Plan 9, and VOS, have been well-integrated into the standard Perl source code kit. You may need to see the *ports/* directory on CPAN for information, and possibly binaries, for the likes of: aos, Atari ST, lynxos, riscos, Novell Netware, Tandem Guardian, *etc.* (Yes, we know that some of these OSes may fall under the Unix category, but we are not a standards body.)

Some approximate operating system names and their `$^O` values in the "OTHER" category include:

OS	<code>\$^O</code>	<code>\$Config{'archname'}</code>

Amiga DOS	amigaos	m68k-amigos
BeOS	beos	
MPE/iX	mpeix	PA-RISC1.1

See also:

- Amiga, *README.amiga* (installed as *perlamiga*).
- Atari, *README.mint* and Guido Flohr's web page <http://stud.uni-sb.de/~guf10000/>
- Be OS, *README.beos*
- HP 300 MPE/iX, *README.mpeix* and Mark Bixby's web page <http://www.bixby.org/mark/perlix.html>
- A free perl5-based PERL.NLM for Novell Netware is available in precompiled binary and source code form from <http://www.novell.com/> as well as from CPAN.
- Plan 9, *README.plan9*

FUNCTION IMPLEMENTATIONS

Listed below are functions that are either completely unimplemented or else have been implemented differently on various platforms. Following each description will be, in parentheses, a list of platforms that the description applies to.

The list may well be incomplete, or even wrong in some places. When in doubt, consult the platform-specific README files in the Perl source distribution, and any other documentation resources accompanying a given port.

Be aware, moreover, that even among Unix-ish systems there are variations.

For many functions, you can also query `%Config`, exported by default from the Config module. For example, to check whether the platform has the `lstat` call, check `$Config{d_lstat}`. See *Config* for a full description of available variables.

Alphabetical Listing of Perl Functions

-X FILEHANDLE

-X EXPR

-X -r, -w, and -x have a limited meaning only; directories and applications are executable, and there are no uid/gid considerations. -o is not supported. (Mac OS)

-r, -w, -x, and -o tell whether the file is accessible, which may not reflect UIC-based file protections. (VMS)

-s returns the size of the data fork, not the total size of data fork plus resource fork. (Mac OS).

-s by name on an open file will return the space reserved on disk, rather than the current extent.

-s on an open filehandle returns the current size. (RISC OS)

-R, -W, -X, -O are indistinguishable from -r, -w, -x, -o. (Mac OS, Win32, VMS, RISC OS)

-b, -c, -k, -g, -p, -u, -A are not implemented. (Mac OS)

-g, -k, -l, -p, -u, -A are not particularly meaningful. (Win32, VMS, RISC OS)

-d is true if passed a device spec without an explicit directory. (VMS)

-T and -B are implemented, but might misclassify Mac text files with foreign characters; this is the case will all platforms, but may affect Mac OS often. (Mac OS)

-x (or -X) determine if a file ends in one of the executable suffixes. -S is meaningless. (Win32)

-x (or -X) determine if a file has an executable file type. (RISC OS)

alarm SECONDS

alarm Not implemented. (Win32)

binmode FILEHANDLE

Meaningless. (Mac OS, RISC OS)

Reopens file and restores pointer; if function fails, underlying filehandle may be closed, or pointer may be in a different position. (VMS)

The value returned by `tell` may be affected after the call, and the filehandle may be flushed. (Win32)

chmod LIST

Only limited meaning. Disabling/enabling write permission is mapped to locking/unlocking the file. (Mac OS)

Only good for changing "owner" read-write access, "group", and "other" bits are meaningless. (Win32)

Only good for changing "owner" and "other" read-write access. (RISC OS)

Access permissions are mapped onto VOS access-control list changes. (VOS)

The actual permissions set depend on the value of the `CYGWIN` in the `SYSTEM` environment settings. (Cygwin)

chown LIST

Not implemented. (Mac OS, Win32, Plan 9, RISC OS, VOS)

Does nothing, but won't fail. (Win32)

chroot FILENAME

chroot Not implemented. (Mac OS, Win32, VMS, Plan 9, RISC OS, VOS, VM/ESA)

crypt PLAINTEXT,SALT

May not be available if library or source was not provided when building perl. (Win32)

Not implemented. (VOS)

dbmclose HASH

Not implemented. (VMS, Plan 9, VOS)

dbmopen HASH,DBNAME,MODE

Not implemented. (VMS, Plan 9, VOS)

dump LABEL

Not useful. (Mac OS, RISC OS)

Not implemented. (Win32)

Invokes VMS debugger. (VMS)

exec LIST

Not implemented. (Mac OS)

Implemented via Spawn. (VM/ESA)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

exit EXPR

exit Emulates UNIX `exit()` (which considers `exit 1` to indicate an error) by mapping the 1 to `SS$_ABORT` (44). This behavior may be overridden with the pragma `use vmsish 'exit'`. As with the CRTL's `exit()` function, `exit 0` is also mapped to an exit status of `SS$_NORMAL` (1); this mapping cannot be overridden. Any other argument to `exit()` is used directly as Perl's exit status. (VMS)

fcntl FILEHANDLE,FUNCTION,SCALAR

Not implemented. (Win32, VMS)

flock FILEHANDLE,OPERATION

Not implemented (Mac OS, VMS, RISC OS, VOS).

Available only on Windows NT (not on Windows 95). (Win32)

fork Not implemented. (Mac OS, AmigaOS, RISC OS, VOS, VM/ESA, VMS)

Emulated using multiple interpreters. See [perlfork](#). (Win32)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

getlogin Not implemented. (Mac OS, RISC OS)

getpgrp PID

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS)

getppid Not implemented. (Mac OS, Win32, RISC OS)

getpriority WHICH,WHO

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS, VM/ESA)

getpwnam NAME

Not implemented. (Mac OS, Win32)

Not useful. (RISC OS)

getgrnam NAME

Not implemented. (Mac OS, Win32, VMS, RISC OS)

getnetbyname NAME

Not implemented. (Mac OS, Win32, Plan 9)

getpwuid UID

Not implemented. (Mac OS, Win32)

Not useful. (RISC OS)

getgrgid GID

Not implemented. (Mac OS, Win32, VMS, RISC OS)

getnetbyaddr ADDR,ADDRTYPE

Not implemented. (Mac OS, Win32, Plan 9)

getprotobynumber NUMBER

Not implemented. (Mac OS)

getservbyport PORT,PROTO

Not implemented. (Mac OS)

getpwent Not implemented. (Mac OS, Win32, VM/ESA)

getgrent Not implemented. (Mac OS, Win32, VMS, VM/ESA)

gethostent

Not implemented. (Mac OS, Win32)

getnetent Not implemented. (Mac OS, Win32, Plan 9)

getprotoent

Not implemented. (Mac OS, Win32, Plan 9)

getservent

Not implemented. (Win32, Plan 9)

sethostent STAYOPEN

Not implemented. (Mac OS, Win32, Plan 9, RISC OS)

setnetent STAYOPEN

Not implemented. (Mac OS, Win32, Plan 9, RISC OS)

setprotoent STAYOPEN

Not implemented. (Mac OS, Win32, Plan 9, RISC OS)

setservent STAYOPEN

Not implemented. (Plan 9, Win32, RISC OS)

endpwent

Not implemented. (Mac OS, MPE/iX, VM/ESA, Win32)

endgrent Not implemented. (Mac OS, MPE/iX, RISC OS, VM/ESA, VMS, Win32)

endhostent

Not implemented. (Mac OS, Win32)

endnetent

Not implemented. (Mac OS, Win32, Plan 9)

endprotoent

Not implemented. (Mac OS, Win32, Plan 9)

endservent

Not implemented. (Plan 9, Win32)

getsockopt SOCKET,LEVEL,OPTNAME

Not implemented. (Plan 9)

glob EXPR

glob This operator is implemented via the File::Glob extension on most platforms. See [File::Glob](#) for portability information.

ioctl FILEHANDLE,FUNCTION,SCALAR

Not implemented. (VMS)

Available only for socket handles, and it does what the `ioctlsocket()` call in the Winsock

API does. (Win32)

Available only for socket handles. (RISC OS)

kill SIGNAL, LIST

`kill(0, LIST)` is implemented for the sake of taint checking; use with other signals is unimplemented. (Mac OS)

Not implemented, hence not useful for taint checking. (RISC OS)

`kill()` doesn't have the semantics of `raise()`, i.e. it doesn't send a signal to the identified process like it does on Unix platforms. Instead `kill($sig, $pid)` terminates the process identified by `$pid`, and makes it exit immediately with exit status `$sig`. As in Unix, if `$sig` is 0 and the specified process exists, it returns true without actually terminating it. (Win32)

link OLDFILE,NEWFILE

Not implemented. (Mac OS, MPE/iX, VMS, RISC OS)

Link count not updated because hard links are not quite that hard (They are sort of half-way between hard and soft links). (AmigaOS)

Hard links are implemented on Win32 (Windows NT and Windows 2000) under NTFS only.

lstat FILEHANDLE

lstat EXPR

`lstat` Not implemented. (VMS, RISC OS)

Return values (especially for device and inode) may be bogus. (Win32)

msgctl ID,CMD,ARG

msgget KEY,FLAGS

msgsnd ID,MSG,FLAGS

msgrcv ID,VAR,SIZE,TYPE,FLAGS

Not implemented. (Mac OS, Win32, VMS, Plan 9, RISC OS, VOS)

open FILEHANDLE,EXPR

open FILEHANDLE

The `|` variants are supported only if ToolServer is installed. (Mac OS)

`open to |` and `- |` are unsupported. (Mac OS, Win32, RISC OS)

Opening a process does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

pipe READHANDLE,WRITEHANDLE

Very limited functionality. (MiNT)

readlink EXPR

`readlink` Not implemented. (Win32, VMS, RISC OS)

select RBITS,WBITS,EBITS,TIMEOUT

Only implemented on sockets. (Win32, VMS)

Only reliable on sockets. (RISC OS)

Note that the `select FILEHANDLE` form is generally portable.

semctl ID,SEMNUM,CMD,ARG

semget KEY,NSEMS,FLAGS

semop KEY,OPSTRING

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS)

setgrent Not implemented. (Mac OS, MPE/iX, VMS, Win32, RISC OS)

setpgrp PID,PGRP

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS)

setpriority WHICH,WHO,PRIORITY

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS)

setpwent Not implemented. (Mac OS, MPE/iX, Win32, RISC OS)

setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL

Not implemented. (Plan 9)

shmctl ID,CMD,ARG

shmget KEY,SIZE,FLAGS

shmread ID,VAR,POS,SIZE

shmwrite ID,STRING,POS,SIZE

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS)

socketatmark SOCKET

A relatively recent addition to socket functions, may not be implemented even in UNIX platforms.

socketpair SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL

Not implemented. (Win32, VMS, RISC OS, VOS, VM/ESA)

stat FILEHANDLE

stat EXPR

stat Platforms that do not have rdev, blksize, or blocks will return these as "", so numeric comparison or manipulation of these fields may cause 'not numeric' warnings.

mtime and atime are the same thing, and ctime is creation time instead of inode change time. (Mac OS).

ctime not supported on UFS (Mac OS X).

ctime is creation time instead of inode change time (Win32).

device and inode are not meaningful. (Win32)

device and inode are not necessarily reliable. (VMS)

mtime, atime and ctime all return the last modification time. Device and inode are not necessarily reliable. (RISC OS)

dev, rdev, blksize, and blocks are not available. inode is not meaningful and will differ between stat calls on the same file. (os2)

some versions of cygwin when doing a stat("foo") and if not finding it may then attempt to stat("foo.exe") (Cygwin)

symlink OLDFILE,NEWFILE

Not implemented. (Win32, VMS, RISC OS)

syscall LIST

Not implemented. (Mac OS, Win32, VMS, RISC OS, VOS, VM/ESA)

sysopen FILEHANDLE,FILENAME,MODE,PERMS

The traditional "0", "1", and "2" MODEs are implemented with different numeric values on some systems. The flags exported by `Fcntl` (`O_RDONLY`, `O_WRONLY`, `O_RDWR`) should work everywhere though. (Mac OS, OS/390, VM/ESA)

system LIST

In general, do not assume the UNIX/POSIX semantics that you can shift `$?` right by eight to get the exit value, or that `$? & 127` would give you the number of the signal that terminated the program, or that `$? & 128` would test true if the program was terminated by a core dump. Instead, use the POSIX `W*` () interfaces: for example, use `WIFEXITED($?)` and `WEXITVALUE($?)` to test for a normal exit and the exit value, `WIFSIGNALED($?)` and `WTERMSIG($?)` for a signal exit and the signal. Core dumping is not a portable concept, so there's no portable way to test for that.

Only implemented if ToolServer is installed. (Mac OS)

As an optimization, may not call the command shell specified in `$ENV{PERL5SHELL}`. `system(1, @args)` spawns an external process and immediately returns its process designator, without waiting for it to terminate. Return value may be used subsequently in `wait` or `waitpid`. Failure to `spawn()` a subprocess is indicated by setting `$?` to "255 << 8". `$?` is set in a way compatible with Unix (i.e. the `exitstatus` of the subprocess is obtained by "`$? >> 8`", as described in the documentation). (Win32)

There is no shell to process metacharacters, and the native standard is to pass a command line terminated by `"\n"` `"\r"` or `"\0"` to the spawned program. Redirection such as `< foo` is performed (if at all) by the run time library of the spawned program. `system list` will call the Unix emulation library's `exec` emulation, which attempts to provide emulation of the `stdin`, `stdout`, `stderr` in force in the parent, providing the child program uses a compatible version of the emulation library. `scalar` will call the native command line direct and no such emulation of a child Unix program will exist. Mileage **will** vary. (RISC OS)

Far from being POSIX compliant. Because there may be no underlying `/bin/sh` tries to work around the problem by forking and `exec`ing the first token in its argument string. Handles basic redirection ("`<`" or "`"`") on its own behalf. (MiNT)

Does not automatically flush output handles on some platforms. (SunOS, Solaris, HP-UX)

The return value is POSIX-like (shifted up by 8 bits), which only allows room for a made-up value derived from the severity bits of the native 32-bit condition code (unless overridden by use `vmsish 'status'`). For more details see [\\$?](#). (VMS)

times Only the first entry returned is nonzero. (Mac OS)

"cumulative" times will be bogus. On anything other than Windows NT or Windows 2000, "system" time will be bogus, and "user" time is actually the time returned by the `clock()` function in the C runtime library. (Win32)

Not useful. (RISC OS)

truncate FILEHANDLE,LENGTH

truncate EXPR,LENGTH

Not implemented. (Older versions of VMS)

Truncation to zero-length only. (VOS)

If a `FILEHANDLE` is supplied, it must be writable and opened in append mode (i.e., use `<< open(FH, 'filename')` or `sysopen(FH, ..., O_APPEND|O_RDWR)`. If a filename is supplied, it should not be held open elsewhere. (Win32)

umask EXPR

umask Returns undef where unavailable, as of version 5.005.

`umask` works but the correct permissions are set only when the file is finally closed. (AmigaOS)

utime LIST

Only the modification time is updated. (BeOS, Mac OS, VMS, RISC OS)

May not behave as expected. Behavior depends on the C runtime library's implementation of `utime()`, and the filesystem being used. The FAT filesystem typically does not support an "access time" field, and it may limit timestamps to a granularity of two seconds. (Win32)

wait**waitpid PID,FLAGS**

Not implemented. (Mac OS, VOS)

Can only be applied to process handles returned for processes spawned using `system(1, . . .)` or pseudo processes created with `fork()`. (Win32)

Not useful. (RISC OS)

CHANGES**v1.48, 02 February 2001**

Various updates from perl5-porters over the past year, supported platforms update from Jarkko Hietaniemi.

v1.47, 22 March 2000

Various cleanups from Tom Christiansen, including migration of long platform listings from [perl](#).

v1.46, 12 February 2000

Updates for VOS and MPE/iX. (Peter Prymmer) Other small changes.

v1.45, 20 December 1999

Small changes from 5.005_63 distribution, more changes to EBCDIC info.

v1.44, 19 July 1999

A bunch of updates from Peter Prymmer for $\O values, endianness, File::Spec, VMS, BS2000, OS/400.

v1.43, 24 May 1999

Added a lot of cleaning up from Tom Christiansen.

v1.42, 22 May 1999

Added notes about tests, `sprintf/printf`, and epoch offsets.

v1.41, 19 May 1999

Lots more little changes to formatting and content.

Added a bunch of $\O and related values for various platforms; fixed mail and web addresses, and added and changed miscellaneous notes. (Peter Prymmer)

v1.40, 11 April 1999

Miscellaneous changes.

v1.39, 11 February 1999

Changes from Jarkko and EMX URL fixes Michael Schwern. Additional note about newlines added.

v1.38, 31 December 1998

More changes from Jarkko.

v1.37, 19 December 1998

More minor changes. Merge two separate version 1.35 documents.

v1.36, 9 September 1998

Updated for Stratus VOS. Also known as version 1.35.

v1.35, 13 August 1998

Integrate more minor changes, plus addition of new sections under *"ISSUES"*:

"Numbers endianness and Width", *"Character sets and character encoding"*, *"Internationalisation"*.

v1.33, 06 August 1998

Integrate more minor changes.

v1.32, 05 August 1998

Integrate more minor changes.

v1.30, 03 August 1998

Major update for RISC OS, other minor changes.

v1.23, 10 July 1998

First public release with perl5.005.

Supported Platforms

As of June 2002 (the Perl release 5.8.0), the following platforms are able to build Perl from the standard source code distribution available at <http://www.cpan.org/src/index.html>

AIX
BeOS
Cygwin
DG/UX
DOS DJGPP 1)
DYNIX/ptx
EPOC R5
FreeBSD
HP-UX
IRIX
Linux
Mac OS Classic
Mac OS X (Darwin)
MPE/iX
NetBSD
NetWare
NonStop-UX
ReliantUNIX (SINIX)
OpenBSD
OpenVMS (VMS)
OS/2
PowerUX
POSIX-BC (BS2000)
QNX
Solaris
SunOS 4
SUPER-UX
Tru64 UNIX (DEC OSF/1, Digital UNIX)
UNICOS
UNICOS/mk
UTS
VOS
Win95/98/ME/2K/XP 2)

WinCE
z/OS (OS/390)
VM/ESA

- 1) in DOS mode either the DOS or OS/2 ports can be used
- 2) compilers: Borland, MinGW (GCC), VC6

The following platforms worked with the previous releases (5.6 and 5.7), but we did not manage either to fix or to test these in time for the 5.8.0 release. There is a very good chance that many of these will work fine with the 5.8.0.

BSD/OS
DomainOS
Hurd
LynxOS
MachTen
PowerMAX
SCO SV
SVR4
Unixware
Windows 3.1

Known to be broken for 5.8.0 (but 5.6.1 and 5.7.2 can be used):

AmigaOS

The following platforms have been known to build Perl from source in the past (5.005_03 and earlier), but we haven't been able to verify their status for the current release, either because the hardware/software platforms are rare or because we don't have an active champion on these platforms—or both. They used to work, though, so go ahead and try compiling them, and let perlbug@perl.org of any trouble.

3b1
A/UX
ConvexOS
CX/UX
DC/OSx
DDE SMES
DOS EMX
Dynix
EP/IX
ESIX
FPS
GENIX
Greenhills
ISC
MachTen 68k
MiNT
MPC
NEWS-OS
NextSTEP
OpenSTEP
Opus
Plan 9
RISC/os
SCO ODT/OSR
Stellar
SVR2
TI1500

TitanOS
 Ultrix
 Unisys Dynix

The following platforms have their own source code distributions and binaries available via <http://www.cpan.org/ports/>

	Perl release
OS/400	5.005_02
Tandem Guardian	5.004

The following platforms have only binaries available via <http://www.cpan.org/ports/index.html> :

	Perl release
Acorn RISCOS	5.005_02
AOS	5.002
LynxOS	5.004_02

Although we do suggest that you always build your own Perl from the source code, both for maximal configurability and for security, in case you are in a hurry you can check <http://www.cpan.org/ports/index.html> for binary distributions.

SEE ALSO

[perlaix](#), [perlamiga](#), [perlapollo](#), [perlbeos](#), [perlbs2000](#), [perlce](#), [perlcygwin](#), [perldgux](#), [perldos](#), [perlepoc](#), [perlebcdic](#), [perlfreesbd](#), [perlhurd](#), [perlhpx](#), [perlirix](#), [perlmachten](#), [perlmacos](#), [perlminit](#), [perlmpaix](#), [perlntware](#), [perlos2](#), [perlos390](#), [perlplan9](#), [perlqnx](#), [perlsolaris](#), [perltru64](#), [perlunicode](#), [perlvmsa](#), [perlvms](#), [perlvos](#), [perlwin32](#), and [Win32](#).

AUTHORS / CONTRIBUTORS

Abigail <abigail@foad.org>, Charles Bailey <bailey@newman.upenn.edu>, Graham Barr <gbarr@pobox.com>, Tom Christiansen <tchrist@perl.com>, Nicholas Clark <nick@ccl4.org>, Thomas Dorner <Thomas.Dorner@start.de>, Andy Dougherty <doughera@lafayette.edu>, Dominic Dunlop <domo@computer.org>, Neale Ferguson <neale@vma.tabnsw.com.au>, David J. Fiander <davidf@mks.com>, Paul Green <Paul_Green@stratus.com>, M.J.T. Guy <mjtg@cam.ac.uk>, Jarkko Hietaniemi <jhi@iki.fi>, Luther Huffman <lutherh@stratcom.com>, Nick Ing-Simmons <nick@ing-simmons.net>, Andreas J. König <a.koenig@mind.de>, Markus Laker <mlaker@contax.co.uk>, Andrew M. Langmead <aml@world.std.com>, Larry Moore <ljmoore@freespace.net>, Paul Moore <Paul.Moore@uk.origin-it.com>, Chris Nandor <pudge@pobox.com>, Matthias Neeracher <neeracher@mac.com>, Philip Newton <pne@cpan.org>, Gary Ng <71564.1743@CompuServe.COM>, Tom Phoenix <rootbeer@teleport.com>, André Pirard <A.Pirard@ulg.ac.be>, Peter Prymmer <pvhp@forte.com>, Hugo van der Sanden <hv@crypt0.demon.co.uk>, Gurusamy Sarathy <gsar@activestate.com>, Paul J. Schinder <schinder@pobox.com>, Michael G Schwern <schwern@pobox.com>, Dan Sugalski <dan@sidhe.org>, Nathan Torkington <gmat@frii.com>.

NAME

perlre – Perl regular expressions

DESCRIPTION

This page describes the syntax of regular expressions in Perl.

if you haven't used regular expressions before, a quick-start introduction is available in [perlrequick](#), and a longer tutorial introduction is available in [perlretut](#).

For reference on how regular expressions are used in matching operations, plus various examples of the same, see discussions of `m//`, `s///`, `qr//` and `??` in [Regex Quote-Like Operators in perlop](#).

Matching operations can have various modifiers. Modifiers that relate to the interpretation of the regular expression inside are listed below. Modifiers that alter the way a regular expression is used by Perl are detailed in [Regex Quote-Like Operators in perlop](#) and [Gory details of parsing quoted constructs in perlop](#).

i Do case-insensitive pattern matching.

If `use locale` is in effect, the case map is taken from the current locale. See [perllocale](#).

m Treat string as multiple lines. That is, change `"^"` and `"$"` from matching the start or end of the string to matching the start or end of any line anywhere within the string.

s Treat string as single line. That is, change `"."` to match any character whatsoever, even a newline, which normally it would not match.

The `/s` and `/m` modifiers both override the `$*` setting. That is, no matter what `$*` contains, `/s` without `/m` will force `"^"` to match only at the beginning of the string and `"$"` to match only at the end (or just before a newline at the end) of the string. Together, as `/ms`, they let the `"."` match any character whatsoever, while still allowing `"^"` and `"$"` to match, respectively, just after and just before newlines within the string.

x Extend your pattern's legibility by permitting whitespace and comments.

These are usually written as "the `/x` modifier", even though the delimiter in question might not really be a slash. Any of these modifiers may also be embedded within the regular expression itself using the `(?...)` construct. See below.

The `/x` modifier itself needs a little more explanation. It tells the regular expression parser to ignore whitespace that is neither backslashed nor within a character class. You can use this to break up your regular expression into (slightly) more readable parts. The `#` character is also treated as a metacharacter introducing a comment, just as in ordinary Perl code. This also means that if you want real whitespace or `#` characters in the pattern (outside a character class, where they are unaffected by `/x`), that you'll either have to escape them or encode them using octal or hex escapes. Taken together, these features go a long way towards making Perl's regular expressions more readable. Note that you have to be careful not to include the pattern delimiter in the comment—perl has no way of knowing you did not intend to close the pattern early. See the C-comment deletion code in [perlop](#).

Regular Expressions

The patterns used in Perl pattern matching derive from supplied in the Version 8 regex routines. (The routines are derived (distantly) from Henry Spencer's freely redistributable reimplementation of the V8 routines.) See [Version 8 Regular Expressions](#) for details.

In particular the following metacharacters have their standard *egrep*-ish meanings:

<code>\</code>	Quote the next metacharacter
<code>^</code>	Match the beginning of the line
<code>.</code>	Match any character (except newline)
<code>\$</code>	Match the end of the line (or before newline at the end)
<code> </code>	Alternation

- () Grouping
- [] Character class

By default, the "^" character is guaranteed to match only the beginning of the string, the "\$" character only the end (or before the newline at the end), and Perl does certain optimizations with the assumption that the string contains only one line. Embedded newlines will not be matched by "^" or "\$". You may, however, wish to treat a string as a multi-line buffer, such that the "^" will match after any newline within the string, and "\$" will match before any newline. At the cost of a little more overhead, you can do this by using the /m modifier on the pattern match operator. (Older programs did this by setting \$*, but this practice is now deprecated.)

To simplify multi-line substitutions, the "." character never matches a newline unless you use the /s modifier, which in effect tells Perl to pretend the string is a single line—even if it isn't. The /s modifier also overrides the setting of \$*, in case you have some (badly behaved) older code that sets it in another module.

The following standard quantifiers are recognized:

- * Match 0 or more times
- + Match 1 or more times
- ? Match 1 or 0 times
- {n} Match exactly n times
- {n,} Match at least n times
- {n,m} Match at least n but not more than m times

(If a curly bracket occurs in any other context, it is treated as a regular character.) The "*" modifier is equivalent to {0,}, the "+" modifier to {1,}, and the "?" modifier to {0,1}. n and m are limited to integral values less than a preset limit defined when perl is built. This is usually 32766 on the most common platforms. The actual limit can be seen in the error message generated by code such as this:

```
$_ **= $_ , / {$_} / for 2 .. 42;
```

By default, a quantified subpattern is "greedy", that is, it will match as many times as possible (given a particular starting location) while still allowing the rest of the pattern to match. If you want it to match the minimum number of times possible, follow the quantifier with a "?". Note that the meanings don't change, just the "greediness":

- *? Match 0 or more times
- +? Match 1 or more times
- ?? Match 0 or 1 time
- {n}? Match exactly n times
- {n,}? Match at least n times
- {n,m}? Match at least n but not more than m times

Because patterns are processed as double quoted strings, the following also work:

- \t tab (HT, TAB)
- \n newline (LF, NL)
- \r return (CR)
- \f form feed (FF)
- \a alarm (bell) (BEL)
- \e escape (think troff) (ESC)
- \033 octal char (think of a PDP-11)
- \x1B hex char
- \x{263a} wide hex char (Unicode SMILEY)
- \c[control char
- \N{name} named char
- \l lowercase next char (think vi)
- \u uppercase next char (think vi)

```

\L      lowercase till \E (think vi)
\U      uppercase till \E (think vi)
\E      end case modification (think vi)
\Q      quote (disable) pattern metacharacters till \E

```

If use `locale` is in effect, the case map used by `\l`, `\L`, `\u` and `\U` is taken from the current locale. See [perllocale](#). For documentation of `\N{name}`, see [charnames](#).

You cannot include a literal `$` or `@` within a `\Q` sequence. An unescaped `$` or `@` interpolates the corresponding variable, while escaping will cause the literal string `\$` to be matched. You'll need to write something like `m/\Quser\E\@\Qhost/`.

In addition, Perl defines the following:

```

\w  Match a "word" character (alphanumeric plus "_")
\W  Match a non-"word" character
\s  Match a whitespace character
\S  Match a non-whitespace character
\d  Match a digit character
\D  Match a non-digit character
\pP Match P, named property. Use \p{Prop} for longer names.
\PP Match non-P
\X  Match eXtended Unicode "combining character sequence",
    equivalent to (?:\PM\pM*)
\C  Match a single C char (octet) even under Unicode.
    NOTE: breaks up characters into their UTF-8 bytes,
    so you may end up with malformed pieces of UTF-8.

```

A `\w` matches a single alphanumeric character (an alphabetic character, or a decimal digit) or `_`, not a whole word. Use `\w+` to match a string of Perl-identifier characters (which isn't the same as matching an English word). If use `locale` is in effect, the list of alphabetic characters generated by `\w` is taken from the current locale. See [perllocale](#). You may use `\w`, `\W`, `\s`, `\S`, `\d`, and `\D` within character classes, but if you try to use them as endpoints of a range, that's not a range, the `"-"` is understood literally. If Unicode is in effect, `\s` matches also `"\x{85}"`, `"\x{2028}"`, and `"\x{2029}"`, see [perlunicode](#) for more details about `\pP`, `\PP`, and `\X`, and [perluniintro](#) about Unicode in general. You can define your own `\p` and `\P` properties, see [perlunicode](#).

The POSIX character class syntax

```
[ :class: ]
```

is also available. The available classes and their backslash equivalents (if available) are as follows:

```

alpha
alnum
ascii
blank           [1]
cntrl
digit           \d
graph
lower
print
punct
space           \s   [2]
upper
word            \w   [3]
xdigit

```

- [1] A GNU extension equivalent to `[\t]`, ‘all horizontal whitespace’.
- [2] Not exactly equivalent to `\s` since the `[[:space:]]` includes also the (very rare) ‘vertical tabulator’, `"\ck"`, `chr(11)`.
- [3] A Perl extension, see above.

For example use `[:upper:]` to match all the uppercase characters. Note that the `[]` are part of the `[: :]` construct, not part of the whole character class. For example:

```
[01[:alpha:]]%
```

matches zero, one, any alphabetic character, and the percentage sign.

The following equivalences to Unicode `\p{ }` constructs and equivalent backslash character classes (if available), will hold:

<code>[: . . . :]</code>	<code>\p{ . . . }</code>	backslash
<code>alpha</code>	<code>IsAlpha</code>	
<code>alnum</code>	<code>IsAlnum</code>	
<code>ascii</code>	<code>IsASCII</code>	
<code>blank</code>	<code>IsSpace</code>	
<code>cntrl</code>	<code>IsCntrl</code>	
<code>digit</code>	<code>IsDigit</code>	<code>\d</code>
<code>graph</code>	<code>IsGraph</code>	
<code>lower</code>	<code>IsLower</code>	
<code>print</code>	<code>IsPrint</code>	
<code>punct</code>	<code>IsPunct</code>	
<code>space</code>	<code>IsSpace</code>	
	<code>IsSpacePerl</code>	<code>\s</code>
<code>upper</code>	<code>IsUpper</code>	
<code>word</code>	<code>IsWord</code>	
<code>xdigit</code>	<code>IsXDigit</code>	

For example `[:lower:]` and `\p{ IsLower }` are equivalent.

If the `utf8` pragma is not used but the `locale` pragma is, the classes correlate with the usual `isalpha(3)` interface (except for ‘word’ and ‘blank’).

The assumedly non-obviously named classes are:

cntrl Any control character. Usually characters that don’t produce output as such but instead control the terminal somehow: for example newline and backspace are control characters. All characters with `ord()` less than 32 are most often classified as control characters (assuming ASCII, the ISO Latin character sets, and Unicode), as is the character with the `ord()` value of 127 (DEL).

graph

Any alphanumeric or punctuation (special) character.

print

Any alphanumeric or punctuation (special) character or the space character.

punct

Any punctuation (special) character.

xdigit

Any hexadecimal digit. Though this may feel silly (`[0–9A–Fa–f]` would work just fine) it is included for completeness.

You can negate the `[:]` character classes by prefixing the class name with a ‘^’. This is a Perl extension. For example:

POSIX traditional Unicode

```
[ : ^ digit : ]      \D      \P{IsDigit}
[ : ^ space : ]      \S      \P{IsSpace}
[ : ^ word : ]       \W      \P{IsWord}
```

Perl respects the POSIX standard in that POSIX character classes are only supported within a character class. The POSIX character classes `[cc.]` and `[=cc=]` are recognized but **not** supported and trying to use them will cause an error.

Perl defines the following zero-width assertions:

```
\b  Match a word boundary
\B  Match a non-(word boundary)
\A  Match only at beginning of string
\Z  Match only at end of string, or before newline at the end
\z  Match only at end of string
\G  Match only at pos() (e.g. at the end-of-match position
    of prior m//g)
```

A word boundary (`\b`) is a spot between two characters that has a `\w` on one side of it and a `\W` on the other side of it (in either order), counting the imaginary characters off the beginning and end of the string as matching a `\W`. (Within character classes `\b` represents backspace rather than a word boundary, just as it normally does in any double-quoted string.) The `\A` and `\Z` are just like `"^"` and `"$"`, except that they won't match multiple times when the `/m` modifier is used, while `"^"` and `"$"` will match at every internal line boundary. To match the actual end of the string and not ignore an optional trailing newline, use `\z`.

The `\G` assertion can be used to chain global matches (using `m//g`), as described in

[Regexp Quote-Like Operators in perlop](#). It is also useful when writing lex-like scanners, when you have several patterns that you want to match against consequent substrings of your string, see the previous reference. The actual location where `\G` will match can also be influenced by using `pos()` as an lvalue: see [pos](#). Currently `\G` is only fully supported when anchored to the start of the pattern; while it is permitted to use it elsewhere, as in `/(?<=\G..)/g`, some such uses (`/.\G/g`, for example) currently cause problems, and it is recommended that you avoid such usage for now.

The bracketing construct `(...)` creates capture buffers. To refer to the digitth buffer use `<digit` within the match. Outside the match use `"$"` instead of `"\"`. (The `<digit` notation works in certain circumstances outside the match. See the warning below about `\1` vs `$1` for details.) Referring back to another part of the match is called a *backreference*.

There is no limit to the number of captured substrings that you may use. However Perl also uses `\10`, `\11`, etc. as aliases for `\010`, `\011`, etc. (Recall that 0 means octal, so `\011` is the character at number 9 in your coded character set; which would be the 10th character, a horizontal tab under ASCII.) Perl resolves this ambiguity by interpreting `\10` as a backreference only if at least 10 left parentheses have opened before it. Likewise `\11` is a backreference only if at least 11 left parentheses have opened before it. And so on. `\1` through `\9` are always interpreted as backreferences.

Examples:

```
s/^( [^ ]* ) * ( [^ ]* ) /$2 $1/;      # swap first two words

if (/(.)\1/) {                          # find first doubled char
    print "'$1' is the first doubled character\n";
}

if (/Time: (..):(..):(..)/) {           # parse out values
    $hours = $1;
    $minutes = $2;
    $seconds = $3;
}
```

Several special variables also refer back to portions of the previous match. `$+` returns whatever the last bracket match matched. `$&` returns the entire matched string. (At one point `$0` did also, but now it returns the name of the program.) `$`` returns everything before the matched string. `$'` returns everything after the matched string. And `$$N` contains whatever was matched by the most-recently closed group (submatch). `$$N` can be used in extended patterns (see below), for example to assign a submatch to a variable.

The numbered variables (`$1`, `$2`, `$3`, etc.) and the related punctuation set (`$+`, `$&`, `$``, `$'`, and `$$N`) are all dynamically scoped until the end of the enclosing block or until the next successful match, whichever comes first. (See [Compound Statements in perlsyn](#).)

WARNING: Once Perl sees that you need one of `$&`, `$``, or `$'` anywhere in the program, it has to provide them for every pattern match. This may substantially slow your program. Perl uses the same mechanism to produce `$1`, `$2`, etc, so you also pay a price for each pattern that contains capturing parentheses. (To avoid this cost while retaining the grouping behaviour, use the extended regular expression `(?: ...)` instead.) But if you never use `$&`, `$`` or `$'`, then patterns *without* capturing parentheses will not be penalized. So avoid `$&`, `$'`, and `$`` if you can, but if you can't (and some algorithms really appreciate them), once you've used them once, use them at will, because you've already paid the price. As of 5.005, `$&` is not so costly as the other two.

Backslashed metacharacters in Perl are alphanumeric, such as `\b`, `\w`, `\n`. Unlike some other regular expression languages, there are no backslashed symbols that aren't alphanumeric. So anything that looks like `\|`, `\(`, `\)`, `\<`, `\`, `\{`, or `\}` is always interpreted as a literal character, not a metacharacter. This was once used in a common idiom to disable or quote the special meanings of regular expression metacharacters in a string that you want to use for a pattern. Simply quote all non-"word" characters:

```
$pattern =~ s/(\\W)/\\$1/g;
```

(If use `locale` is set, then this depends on the current locale.) Today it is more common to use the `quotemeta()` function or the `\Q` metaquoting escape sequence to disable all metacharacters' special meanings like this:

```
/$unquoted\Q$quoted\E$unquoted/
```

Beware that if you put literal backslashes (those not inside interpolated variables) between `\Q` and `\E`, double-quotish backslash interpolation may lead to confusing results. If you *need* to use literal backslashes within `\Q... \E`, consult [Gory details of parsing quoted constructs in perlop](#).

Extended Patterns

Perl also defines a consistent extension syntax for features not found in standard tools like **awk** and **lex**. The syntax is a pair of parentheses with a question mark as the first thing within the parentheses. The character after the question mark indicates the extension.

The stability of these extensions varies widely. Some have been part of the core language for many years. Others are experimental and may change without warning or be completely removed. Check the documentation on an individual feature to verify its current status.

A question mark was chosen for this and for the minimal-matching construct because 1) question marks are rare in older regular expressions, and 2) whenever you see one, you should stop and "question" exactly what is going on. That's psychology...

(`?#text`) A comment. The text is ignored. If the `/x` modifier enables whitespace formatting, a simple `#` will suffice. Note that Perl closes the comment as soon as it sees a `)`, so there is no way to put a literal `)` in the comment.

(`?imsx-imsx`)

One or more embedded pattern-match modifiers, to be turned on (or turned off, if preceded by `-`) for the remainder of the pattern or the remainder of the enclosing pattern group (if any). This is particularly useful for dynamic patterns, such as those read in from a configuration file, read in as an argument, are specified in a table somewhere, etc. Consider the case that some of which want to be case sensitive and some do not. The case insensitive ones need to

include merely `(?i)` at the front of the pattern. For example:

```
$pattern = "foobar";
if ( /$pattern/i ) { }

# more flexible:

$pattern = "(?i)foobar";
if ( /$pattern/ ) { }
```

These modifiers are restored at the end of the enclosing group. For example,

```
( (?i) blah ) \s+ \1
```

will match a repeated (*including the case!*) word `blah` in any case, assuming `x` modifier, and no `i` modifier outside this group.

`(?:pattern)`

`(?imsx-imsx:pattern)`

This is for clustering, not capturing; it groups subexpressions like `"()"`, but doesn't make backreferences as `"()"` does. So

```
@fields = split(/\b(?:a|b|c)\b/)
```

is like

```
@fields = split(/\b(a|b|c)\b/)
```

but doesn't spit out extra fields. It's also cheaper not to capture characters if you don't need to.

Any letters between `?` and `:` act as flags modifiers as with `(?imsx-imsx)`. For example,

```
/(?:s-i:more.*than).*million/i
```

is equivalent to the more verbose

```
/(?: (?:s-i)more.*than).*million/i
```

`(?=pattern)`

A zero-width positive look-ahead assertion. For example, `/\w+(?=\t)/` matches a word followed by a tab, without including the tab in `$&`.

`(?!pattern)`

A zero-width negative look-ahead assertion. For example `/foo(?!bar)/` matches any occurrence of "foo" that isn't followed by "bar". Note however that look-ahead and look-behind are NOT the same thing. You cannot use this for look-behind.

If you are looking for a "bar" that isn't preceded by a "foo", `/(?!foo)bar/` will not do what you want. That's because the `(?!foo)` is just saying that the next thing cannot be "foo"—and it's not, it's a "bar", so "foobar" will match. You would have to do something like `/(?!foo)\.bar/` for that. We say "like" because there's the case of your "bar" not having three characters before it. You could cover that this way:

`/(?: (?!foo) \. | ^.{0,2}) bar/`. Sometimes it's still easier just to say:

```
if (/bar/ && $` !~ /foo$/)
```

For look-behind see below.

`(?<=pattern)`

A zero-width positive look-behind assertion. For example, `/(?<=\t)\w+/` matches a word that follows a tab, without including the tab in `$&`. Works only for fixed-width look-behind.

(?<!pattern)

A zero-width negative look-behind assertion. For example `/ (?<!bar) foo /` matches any occurrence of "foo" that does not follow "bar". Works only for fixed-width look-behind.

(?{ code })

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice.

This zero-width assertion evaluate any embedded Perl code. It always succeeds, and its code is not interpolated. Currently, the rules to determine where the `code` ends are somewhat convoluted.

This feature can be used together with the special variable `$$N` to capture the results of submatches in variables without having to keep track of the number of nested parentheses. For example:

```
$_ = "The brown fox jumps over the lazy dog";
/the (\S+) (?{ $color = $$N }) (\S+) (?{ $animal = $$N })/i;
print "color = $color, animal = $animal\n";
```

The code is properly scoped in the following sense: If the assertion is backtracked (compare *"Backtracking"*), all changes introduced after localization are undone, so that

```
$_ = 'a' x 8;
m<
    (?{ $cnt = 0 })                # Initialize $cnt.
    (
        a
        (?{
            local $cnt = $cnt + 1;    # Update $cnt, backtracking-safe.
        })
    ) *
aaaa
    (?{ $res = $cnt })              # On success copy to non-localized
                                    # location.
>x;
```

will set `$res = 4`. Note that after the match, `$cnt` returns to the globally introduced value, because the scopes that restrict local operators are unwound.

This assertion may be used as a `(?(condition)yes-pattern|no-pattern)` switch.

If *not* used in this way, the result of evaluation of `code` is put into the special variable `$$R`. This happens immediately, so `$$R` can be used from other `(?{ code })` assertions inside the same regular expression.

The assignment to `$$R` above is properly localized, so the old value of `$$R` is restored if the assertion is backtracked; compare *"Backtracking"*.

For reasons of security, this construct is forbidden if the regular expression involves run-time interpolation of variables, unless the perilous use `re 'eval'` pragma has been used (see *re*), or the variables contain results of `qr//` operator (see *qr/STRING/imosx in perlop*).

This restriction is because of the wide-spread and remarkably convenient custom of using run-time determined strings as patterns. For example:

```
$re = <>;
chomp $re;
$string =~ /$re/;
```

Before Perl knew how to execute interpolated code within a pattern, this operation was

completely safe from a security point of view, although it could raise an exception from an illegal pattern. If you turn on the use `re 'eval'`, though, it is no longer secure, so you should only do so if you are also using taint checking. Better yet, use the carefully constrained evaluation within a Safe module. See [perlsec](#) for details about both these mechanisms.

`(??{ code })`

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice. A simplified version of the syntax may be introduced for commonly used idioms.

This is a "postponed" regular subexpression. The `code` is evaluated at run time, at the moment this subexpression may match. The result of evaluation is considered as a regular expression and matched as if it were inserted instead of this construct.

The `code` is not interpolated. As before, the rules to determine where the `code` ends are currently somewhat convoluted.

The following pattern matches a parenthesized group:

```
$re = qr{
    \ (
      (? :
        (?> [^()]+ )      # Non-parens without backtracking
        |
        (??{ $re })      # Group with matching parens
      ) *
    \ )
  } x;
```

`< (?pattern)`

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice.

An "independent" subexpression, one which matches the substring that a *standalone pattern* would match if anchored at the given position, and it matches *nothing other than this substring*. This construct is useful for optimizations of what would otherwise be "eternal" matches, because it will not backtrack (see ["Backtracking"](#)). It may also be useful in places where the "grab all you can, and do not give anything back" semantic is desirable.

For example: `< ^(?a*)ab` will never match, since `< (?a*)` (anchored at the beginning of string, as above) will match *all* characters `a` at the beginning of string, leaving no `a` for `ab` to match. In contrast, `a*ab` will match the same as `a+b`, since the match of the subgroup `a*` is influenced by the following group `ab` (see ["Backtracking"](#)). In particular, `a*` inside `a*ab` will match fewer characters than a standalone `a*`, since this makes the tail match.

An effect similar to `< (?pattern)` may be achieved by writing `(?= (pattern)) \1`. This matches the same substring as a standalone `a+`, and the following `\1` eats the matched string; it therefore makes a zero-length assertion into an analogue of `< (?...)`. (The difference between these two constructs is that the second one uses a capturing group, thus shifting ordinals of backreferences in the rest of a regular expression.)

Consider this pattern:

```
m{ \ (
    (
      [^()]+          # x+
    |
      \ ( [^()] * \ )
    ) +
  \ )
```

```
}x
```

That will efficiently match a nonempty group with matching parentheses two levels deep or less. However, if there is no such group, it will take virtually forever on a long string. That's because there are so many different ways to split a long string into several substrings. This is what `(.+) +` is doing, and `(.+) +` is similar to a subpattern of the above pattern. Consider how the pattern above detects no-match on `((()aaaaaaaaaaaaaaaaaaaaa` in several seconds, but that each extra letter doubles this time. This exponential performance will make it appear that your program has hung. However, a tiny change to this pattern

```
m{ \ (
    (
        (?> [^()]+ )          # change x+ above to (?> x+ )
        |
        \ ( [^()] * \ )
    ) +
    \ )
}x
```

which uses `< (?...)` matches exactly when the one above does (verifying this yourself would be a productive exercise), but finishes in a fourth the time when used on a similar string with 1000000 as. Be aware, however, that this pattern currently triggers a warning message under the use `warnings` pragma or `-w` switch saying it "matches null string many times in regex".

On simple groups, such as the pattern `< (? [^()] +)`, a comparable effect may be achieved by negative look-ahead, as in `[^()] + (?! [^()])`. This was only 4 times slower on a string with 1000000 as.

The "grab all you can, and do not give anything back" semantic is desirable in many situations where on the first sight a simple `() *` looks like the correct solution. Suppose we parse text with comments being delimited by `#` followed by some optional (horizontal) whitespace. Contrary to its appearance, `# [\t] *` *is not* the correct subexpression to match the comment delimiter, because it may "give up" some whitespace if the remainder of the pattern can be made to match that way. The correct answer is either one of these:

```
(?># [ \t] *)
# [ \t] * (?! [ \t] )
```

For example, to grab non-empty comments into `$1`, one should use either one of these:

```
/ (?> \# [ \t] * ) (      .+ ) /x;
/      \# [ \t] *      ( [^ \t] .* ) /x;
```

Which one you pick depends on which of these expressions better reflects the above specification of comments.

```
(?(condition)yes-pattern|no-pattern)
(?(condition)yes-pattern)
```

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice.

Conditional expression. `(condition)` should be either an integer in parentheses (which is valid if the corresponding pair of parentheses matched), or look-ahead/look-behind/evaluate zero-width assertion.

For example:

```
m{ ( \ ( ) ?
    [ ^ ( ] +
    ( ? ( 1 ) \ ) )
```

}x

matches a chunk of non-parentheses, possibly included in parentheses themselves.

Backtracking

NOTE: This section presents an abstract approximation of regular expression behavior. For a more rigorous (and complicated) view of the rules involved in selecting a match among possible alternatives, see [Combining pieces together](#).

A fundamental feature of regular expression matching involves the notion called *backtracking*, which is currently used (when needed) by all regular expression quantifiers, namely `*`, `*?`, `+`, `++?`, `{n,m}`, and `{n,m}?`. Backtracking is often optimized internally, but the general principle outlined here is valid.

For a regular expression to match, the *entire* regular expression must match, not just part of it. So if the beginning of a pattern containing a quantifier succeeds in a way that causes later parts in the pattern to fail, the matching engine backs up and recalculates the beginning part—that's why it's called backtracking.

Here is an example of backtracking: Let's say you want to find the word following "foo" in the string "Food is on the foo table.":

```
$_ = "Food is on the foo table.";
if ( /\b(foo)\s+(\w+)/i ) {
    print "$2 follows $1.\n";
}
```

When the match runs, the first part of the regular expression (`\b(foo)`) finds a possible match right at the beginning of the string, and loads up `$1` with "Foo". However, as soon as the matching engine sees that there's no whitespace following the "Foo" that it had saved in `$1`, it realizes its mistake and starts over again one character after where it had the tentative match. This time it goes all the way until the next occurrence of "foo". The complete regular expression matches this time, and you get the expected output of "table follows foo."

Sometimes minimal matching can help a lot. Imagine you'd like to match everything between "foo" and "bar". Initially, you write something like this:

```
$_ = "The food is under the bar in the barn.";
if ( /foo(.*?)bar/ ) {
    print "got <$1>\n";
}
```

Which perhaps unexpectedly yields:

```
got <d is under the bar in the >
```

That's because `.*` was greedy, so you get everything between the *first* "foo" and the *last* "bar". Here it's more effective to use minimal matching to make sure you get the text between a "foo" and the first "bar" thereafter.

```
if ( /foo(.*?)bar/ ) { print "got <$1>\n" }
got <d is under the >
```

Here's another example: let's say you'd like to match a number at the end of a string, and you also want to keep the preceding part of the match. So you write this:

```
$_ = "I have 2 numbers: 53147";
if ( /(.*)(\d+)/ ) {                                     # Wrong!
    print "Beginning is <$1>, number is <$2>.\n";
}
```

That won't work at all, because `.*` was greedy and gobbled up the whole string. As `\d*` can match on an empty string the complete regular expression matched successfully.

Beginning is <I have 2 numbers: 53147>, number is <>.

Here are some variants, most of which don't work:

```
$_ = "I have 2 numbers: 53147";
@pats = qw{
    (.*)(\d*)
    (.*)(\d+)
    (.*?)(\d*)
    (.*?)(\d+)
    (.*)(\d+)$
    (.*?)(\d+)$
    (.*)\b(\d+)$
    (.*\D)(\d+)$
};

for $pat (@pats) {
    printf "%-12s ", $pat;
    if ( /$pat/ ) {
        print "<$1> <$2>\n";
    } else {
        print "FAIL\n";
    }
}
```

That will print out:

```
(.)(\d)      <I have 2 numbers: 53147> <>
(.)(\d+)     <I have 2 numbers: 5314> <7>
(.*?)(\d)    <> <>
(.*?)(\d+)   <I have > <2>
(.*)(\d+)$   <I have 2 numbers: 5314> <7>
(.*?)(\d+)$  <I have 2 numbers: > <53147>
(.*)\b(\d+)$ <I have 2 numbers: > <53147>
(.*\D)(\d+)$ <I have 2 numbers: > <53147>
```

As you see, this can be a bit tricky. It's important to realize that a regular expression is merely a set of assertions that gives a definition of success. There may be 0, 1, or several different ways that the definition might succeed against a particular string. And if there are multiple ways it might succeed, you need to understand backtracking to know which variety of success you will achieve.

When using look-ahead assertions and negations, this can all get even trickier. Imagine you'd like to find a sequence of non-digits not followed by "123". You might try to write that as

```
$_ = "ABC123";
if ( /^ \D* (?!123) / ) {           # Wrong!
    print "Yup, no 123 in $_\n";
}
```

But that isn't going to match; at least, not the way you're hoping. It claims that there is no 123 in the string. Here's a clearer picture of why that pattern matches, contrary to popular expectations:

```
$x = 'ABC123' ;
$y = 'ABC445' ;

print "1: got $1\n" if $x =~ /^(ABC) (?!123)/ ;
print "2: got $1\n" if $y =~ /^(ABC) (?!123)/ ;

print "3: got $1\n" if $x =~ /^(\D*) (?!123)/ ;
print "4: got $1\n" if $y =~ /^(\D*) (?!123)/ ;
```

This prints

```
2: got ABC
3: got AB
4: got ABC
```

You might have expected test 3 to fail because it seems to a more general purpose version of test 1. The important difference between them is that test 3 contains a quantifier (`\D*`) and so can use backtracking, whereas test 1 will not. What's happening is that you've asked "Is it true that at the start of `$x`, following 0 or more non-digits, you have something that's not 123?" If the pattern matcher had let `\D*` expand to "ABC", this would have caused the whole pattern to fail.

The search engine will initially match `\D*` with "ABC". Then it will try to match `(?!123` with "123", which fails. But because a quantifier (`\D*`) has been used in the regular expression, the search engine can backtrack and retry the match differently in the hope of matching the complete regular expression.

The pattern really, *really* wants to succeed, so it uses the standard pattern back-off-and-retry and lets `\D*` expand to just "AB" this time. Now there's indeed something following "AB" that is not "123". It's "C123", which suffices.

We can deal with this by using both an assertion and a negation. We'll say that the first part in `$1` must be followed both by a digit and by something that's not "123". Remember that the look-aheads are zero-width expressions—they only look, but don't consume any of the string in their match. So rewriting this way produces what you'd expect; that is, case 5 will fail, but case 6 succeeds:

```
print "5: got $1\n" if $x =~ /^( \D* ) ( ? = \d ) ( ? ! 123 ) / ;
print "6: got $1\n" if $y =~ /^( \D* ) ( ? = \d ) ( ? ! 123 ) / ;

6: got ABC
```

In other words, the two zero-width assertions next to each other work as though they're ANDed together, just as you'd use any built-in assertions: `/^$/` matches only if you're at the beginning of the line AND the end of the line simultaneously. The deeper underlying truth is that juxtaposition in regular expressions always means AND, except when you write an explicit OR using the vertical bar. `/ab/` means match "a" AND (then) match "b", although the attempted matches are made at different positions because "a" is not a zero-width assertion, but a one-width assertion.

WARNING: particularly complicated regular expressions can take exponential time to solve because of the immense number of possible ways they can use backtracking to try match. For example, without internal optimizations done by the regular expression engine, this will take a painfully long time to run:

```
'aaaaaaaaaaaa' =~ / ( ( a { 0 , 5 } ) { 0 , 5 } ) * [ c ] /
```

And if you used `*`'s in the internal groups instead of limiting them to 0 through 5 matches, then it would take forever—or until you ran out of stack space. Moreover, these internal optimizations are not always applicable. For example, if you put `{0,5}` instead of `*` on the external group, no current optimization is applicable, and the match takes a long time to finish.

A powerful tool for optimizing such beasts is what is known as an "independent group", which does not backtrack (see < [\(?pattern\)](#)). Note also that zero-length look-ahead/look-behind assertions will not backtrack to make the tail match, since they are in "logical" context: only whether they match is considered relevant. For an example where side-effects of look-ahead *might* have influenced the following match, see < [\(?pattern\)](#)).

Version 8 Regular Expressions

In case you're not familiar with the "regular" Version 8 regex routines, here are the pattern-matching rules not described above.

Any single character matches itself, unless it is a *metacharacter* with a special meaning described here or above. You can cause characters that normally function as metacharacters to be interpreted literally by prefixing them with a `"\"` (e.g., `"\"` matches a `"`, not any character; `"\"` matches a `"`). A series of

characters matches that series of characters in the target string, so the pattern `blurf1` would match "blurf1" in the target string.

You can specify a character class, by enclosing a list of characters in `[]`, which will match any one character from the list. If the first character after the "[" is "^", the class matches any character not in the list. Within a list, the "-" character specifies a range, so that `a-z` represents all characters between "a" and "z", inclusive. If you want either "-" or "]" itself to be a member of a class, put it at the start of the list (possibly after a "^"), or escape it with a backslash. "-" is also taken literally when it is at the end of the list, just before the closing "]". (The following all specify the same class of three characters: `[-az]`, `[az-]`, and `[a\ -z]`. All are different from `[a-z]`, which specifies a class containing twenty-six characters, even on EBCDIC based coded character sets.) Also, if you try to use the character classes `\w`, `\W`, `\s`, `\S`, `\d`, or `\D` as endpoints of a range, that's not a range, the "-" is understood literally.

Note also that the whole range idea is rather unportable between character sets—and even within character sets they may cause results you probably didn't expect. A sound principle is to use only ranges that begin from and end at either alphabets of equal case (`[a-e]`, `[A-E]`), or digits (`[0-9]`). Anything else is unsafe. If in doubt, spell out the character sets in full.

Characters may be specified using a metacharacter syntax much like that used in C: `"\n"` matches a newline, `"\t"` a tab, `"\r"` a carriage return, `"\f"` a form feed, etc. More generally, `\nnn`, where `nnn` is a string of octal digits, matches the character whose coded character set value is `nnn`. Similarly, `\xnn`, where `nn` are hexadecimal digits, matches the character whose numeric value is `nn`. The expression `\cx` matches the character control-`x`. Finally, the `"."` metacharacter matches any character except `"\n"` (unless you use `/s`).

You can specify a series of alternatives for a pattern using `|` to separate them, so that `fee|fie|foe` will match any of "fee", "fie", or "foe" in the target string (as would `f(e|i|o)e`). The first alternative includes everything from the last pattern delimiter ("`|`", "`[`", or the beginning of the pattern) up to the first "`|`", and the last alternative contains everything from the last "`|`" to the next pattern delimiter. That's why it's common practice to include alternatives in parentheses: to minimize confusion about where they start and end.

Alternatives are tried from left to right, so the first alternative found for which the entire expression matches, is the one that is chosen. This means that alternatives are not necessarily greedy. For example: when matching `foo|foot` against "barefoot", only the "foo" part will match, as that is the first alternative tried, and it successfully matches the target string. (This might not seem important, but it is important when you are capturing matched text using parentheses.)

Also remember that `|` is interpreted as a literal within square brackets, so if you write `[fee|fie|foe]` you're really only matching `[feio|]`.

Within a pattern, you may designate subpatterns for later reference by enclosing them in parentheses, and you may refer back to the *n*th subpattern later in the pattern using the metacharacter `\n`. Subpatterns are numbered based on the left to right order of their opening parenthesis. A backreference matches whatever actually matched the subpattern in the string being examined, not the rules for that subpattern. Therefore, `(0|0x)\d*\s\d*\d*` will match "0x1234 0x4321", but not "0x1234 01234", because subpattern 1 matched "0x", even though the rule `0|0x` could potentially match the leading 0 in the second number.

Warning on \1 vs \$1

Some people get too used to writing things like:

```
$pattern =~ s/(\W)/\\1/g;
```

This is grandfathered for the RHS of a substitute to avoid shocking the **sed** addicts, but it's a dirty habit to get into. That's because in PerlThink, the righthand side of an `s///` is a double-quoted string. `\1` in the usual double-quoted string means a control-A. The customary Unix meaning of `\1` is kludged in for `s///`.

However, if you get into the habit of doing that, you get yourself into trouble if you then add an `/e` modifier.

```
s/(\d+)/ \1 + 1 /eg;          # causes warning under -w
```

Or if you try to do

```
s/(\d+)/\1000/;
```

You can't disambiguate that by saying `\{1\}000`, whereas you can fix it with `\{1\}000`. The operation of interpolation should not be confused with the operation of matching a backreference. Certainly they mean two different things on the *left* side of the `s///`.

Repeated patterns matching zero-length substring

WARNING: Difficult material (and prose) ahead. This section needs a rewrite.

Regular expressions provide a terse and powerful programming language. As with most other power tools, power comes together with the ability to wreak havoc.

A common abuse of this power stems from the ability to make infinite loops using regular expressions, with something as innocuous as:

```
'foo' =~ m{ ( o? ) * }x;
```

The `o?` can match at the beginning of `'foo'`, and since the position in the string is not moved by the match, `o?` would match again and again because of the `*` modifier. Another common way to create a similar cycle is with the looping modifier `/g`:

```
@matches = ( 'foo' =~ m{ o? }xg );
```

or

```
print "match: <$&>\n" while 'foo' =~ m{ o? }xg;
```

or the loop implied by `split()`.

However, long experience has shown that many programming tasks may be significantly simplified by using repeated subexpressions that may match zero-length substrings. Here's a simple example being:

```
@chars = split //, $string;          # // is not magic in split
($whitewashed = $string) =~ s/()/ /g; # parens avoid magic s// /
```

Thus Perl allows such constructs, by *forcefully breaking the infinite loop*. The rules for this are different for lower-level loops given by the greedy modifiers `*+{ }`, and for higher-level ones like the `/g` modifier or `split()` operator.

The lower-level loops are *interrupted* (that is, the loop is broken) when Perl detects that a repeated expression matched a zero-length substring. Thus

```
m{ (?: NON_ZERO_LENGTH | ZERO_LENGTH ) * }x;
```

is made equivalent to

```
m{
    (?: NON_ZERO_LENGTH ) *
    |
    (?: ZERO_LENGTH ) ?
}x;
```

The higher level-loops preserve an additional state between iterations: whether the last match was zero-length. To break the loop, the following match after a zero-length match is prohibited to have a length of zero. This prohibition interacts with backtracking (see "[Backtracking](#)"), and so the *second best* match is chosen if the *best* match is of zero length.

For example:

```
$_ = 'bar';
s/\w??/<$&>/g;
```

results in `< <b<<a<<r<`. At each position of the string the best match given by non-greedy `??` is the zero-length match, and the *second best* match is what is matched by `\w`. Thus zero-length matches alternate with one-character-long matches.

Similarly, for repeated `m/ () /g` the second-best match is the match at the position one notch further in the string.

The additional state of being *matched with zero-length* is associated with the matched string, and is reset by each assignment to `pos()`. Zero-length matches at the end of the previous match are ignored during `split`.

Combining pieces together

Each of the elementary pieces of regular expressions which were described before (such as `ab` or `\Z`) could match at most one substring at the given position of the input string. However, in a typical regular expression these elementary pieces are combined into more complicated patterns using combining operators `ST`, `S|T`, `S*` etc (in these examples `S` and `T` are regular subexpressions).

Such combinations can include alternatives, leading to a problem of choice: if we match a regular expression `a|ab` against `"abc"`, will it match substring `"a"` or `"ab"`? One way to describe which substring is actually matched is the concept of backtracking (see ["Backtracking"](#)). However, this description is too low-level and makes you think in terms of a particular implementation.

Another description starts with notions of "better"/"worse". All the substrings which may be matched by the given regular expression can be sorted from the "best" match to the "worst" match, and it is the "best" match which is chosen. This substitutes the question of "what is chosen?" by the question of "which matches are better, and which are worse?".

Again, for elementary pieces there is no such question, since at most one match at a given position is possible. This section describes the notion of better/worse for combining operators. In the description below `S` and `T` are regular subexpressions.

ST Consider two possible matches, `AB` and `A'B'`, `A` and `A'` are substrings which can be matched by `S`, `B` and `B'` are substrings which can be matched by `T`.

If `A` is better match for `S` than `A'`, `AB` is a better match than `A'B'`.

If `A` and `A'` coincide: `AB` is a better match than `AB'` if `B` is better match for `T` than `B'`.

S|T When `S` can match, it is a better match than when only `T` can match.

Ordering of two matches for `S` is the same as for `S`. Similar for two matches for `T`.

S{REPEAT_COUNT}

Matches as `SSS...S` (repeated as many times as necessary).

S{min,max}

Matches as `S{max}|S{max-1}|...|S{min+1}|S{min}`.

S{min,max}?

Matches as `S{min}|S{min+1}|...|S{max-1}|S{max}`.

S?, S*, S+

Same as `S{0,1}`, `S{0,BIG_NUMBER}`, `S{1,BIG_NUMBER}` respectively.

S??, S*?, S+?

Same as `S{0,1}?`, `S{0,BIG_NUMBER}?`, `S{1,BIG_NUMBER}?` respectively.

< (?S)

Matches the best match for `S` and only that.

(?=S), (?<=S)

Only the best match for `S` is considered. (This is important only if `S` has capturing parentheses, and backreferences are used somewhere else in the whole regular expression.)

(?!S), (?<!S)

For this grouping operator there is no need to describe the ordering, since only whether or not *S* can match is important.

(??{ EXPR })

The ordering is the same as for the regular expression which is the result of *EXPR*.

(?(condition)yes-pattern|no-pattern)

Recall that which of *yes-pattern* or *no-pattern* actually matches is already determined. The ordering of the matches is the same as for the chosen subexpression.

The above recipes describe the ordering of matches *at a given position*. One more rule is needed to understand how a match is determined for the whole regular expression: a match at an earlier position is always better than a match at a later position.

Creating custom RE engines

Overloaded constants (see [overload](#)) provide a simple way to extend the functionality of the RE engine.

Suppose that we want to enable a new RE escape-sequence `\Y|` which matches at boundary between white-space characters and non-whitespace characters. Note that `(?=\S) (?<!\S) | (?!\S) (?<=\S)` matches exactly at these positions, so we want to have each `\Y|` in the place of the more complicated version. We can create a module `customre` to do this:

```
package customre;
use overload;

sub import {
    shift;
    die "No argument to customre::import allowed" if @_;
    overload::constant 'qr' => \&convert;
}

sub invalid { die "/$_[0]/: invalid escape '\$_[1]'" }

my %rules = ( '\\ ' => '\\ ',
              'Y| ' => qr/(?=\S) (?<!\S) | (?!\S) (?<=\S)/ );

sub convert {
    my $re = shift;
    $re =~ s{
        \\ ( \\ | Y . )
    }
        { $rules{$1} or invalid($re,$1) }sgex;
    return $re;
}
```

Now use `customre` enables the new escape in constant regular expressions, i.e., those without any runtime variable interpolations. As documented in [overload](#), this conversion will work only over literal parts of regular expressions. For `\Y|$re\Y|` the variable part of this regular expression needs to be converted explicitly (but only if the special meaning of `\Y|` should be enabled inside `$re`) :

```
use customre;
$re = <>;
chomp $re;
$re = customre::convert $re;
/\Y|$re\Y|/;
```

BUGS

This document varies from difficult to understand to completely and utterly opaque. The wandering prose riddled with jargon is hard to fathom in several places.

This document needs a rewrite that separates the tutorial content from the reference content.

SEE ALSO

[perlrequick](#).

[perlretut](#).

[Regex Quote-Like Operators in perlop](#).

[Gory details of parsing quoted constructs in perlop](#).

[perlfaq6](#).

[pos](#).

[perllocale](#).

[perlebcdic](#).

Mastering Regular Expressions by Jeffrey Friedl, published by O'Reilly and Associates.

NAME

perlref – Perl references and nested data structures

NOTE

This is complete documentation about all aspects of references. For a shorter, tutorial introduction to just the essential features, see [perlreftut](#).

DESCRIPTION

Before release 5 of Perl it was difficult to represent complex data structures, because all references had to be symbolic—and even then it was difficult to refer to a variable instead of a symbol table entry. Perl now not only makes it easier to use symbolic references to variables, but also lets you have "hard" references to any piece of data or code. Any scalar may hold a hard reference. Because arrays and hashes contain scalars, you can now easily build arrays of arrays, arrays of hashes, hashes of arrays, arrays of hashes of functions, and so on.

Hard references are smart—they keep track of reference counts for you, automatically freeing the thing referred to when its reference count goes to zero. (Reference counts for values in self-referential or cyclic data structures may not go to zero without a little help; see [Two-Phased Garbage Collection in perlobj](#) for a detailed explanation.) If that thing happens to be an object, the object is destructed. See [perlobj](#) for more about objects. (In a sense, everything in Perl is an object, but we usually reserve the word for references to objects that have been officially "blessed" into a class package.)

Symbolic references are names of variables or other objects, just as a symbolic link in a Unix filesystem contains merely the name of a file. The `*glob` notation is something of a symbolic reference. (Symbolic references are sometimes called "soft references", but please don't call them that; references are confusing enough without useless synonyms.)

In contrast, hard references are more like hard links in a Unix file system: They are used to access an underlying object without concern for what its (other) name is. When the word "reference" is used without an adjective, as in the following paragraph, it is usually talking about a hard reference.

References are easy to use in Perl. There is just one overriding principle: Perl does no implicit referencing or dereferencing. When a scalar is holding a reference, it always behaves as a simple scalar. It doesn't magically start being an array or hash or subroutine; you have to tell it explicitly to do so, by dereferencing it.

Making References

References can be created in several ways.

1. By using the backslash operator on a variable, subroutine, or value. (This works much like the `&` (address-of) operator in C.) This typically creates *another* reference to a variable, because there's already a reference to the variable in the symbol table. But the symbol table reference might go away, and you'll still have the reference that the backslash returned. Here are some examples:

```
$scalarref = \ $foo;
$arrayref  = \@ARGV;
$hashref   = \%ENV;
$coderef   = \&handler;
$globref   = \*foo;
```

It isn't possible to create a true reference to an IO handle (filehandle or dirhandle) using the backslash operator. The most you can get is a reference to a typeglob, which is actually a complete symbol table entry. But see the explanation of the `*foo{THING}` syntax below. However, you can still use typeglobs and globrefs as though they were IO handles.

2. A reference to an anonymous array can be created using square brackets:

```
$arrayref = [1, 2, ['a', 'b', 'c']];
```

Here we've created a reference to an anonymous array of three elements whose final element is itself a reference to another anonymous array of three elements. (The multidimensional syntax described later can be used to access this. For example, after the above, `< $arrayref->[2][1]` would have the value "b".)

Taking a reference to an enumerated list is not the same as using square brackets—instead it's the same as creating a list of references!

```
@list = (\$a, \@b, \%c);
@list = \($a, @b, %c);      # same thing!
```

As a special case, `\(@foo)` returns a list of references to the contents of `@foo`, not a reference to `@foo` itself. Likewise for `%foo`, except that the key references are to copies (since the keys are just strings rather than full-fledged scalars).

3. A reference to an anonymous hash can be created using curly brackets:

```
$hashref = {
    'Adam'   => 'Eve',
    'Clyde'  => 'Bonnie',
};
```

Anonymous hash and array composers like these can be intermixed freely to produce as complicated a structure as you want. The multidimensional syntax described below works for these too. The values above are literals, but variables and expressions would work just as well, because assignment operators in Perl (even within `local()` or `my()`) are executable statements, not compile-time declarations.

Because curly brackets (braces) are used for several other things including BLOCKs, you may occasionally have to disambiguate braces at the beginning of a statement by putting a `+` or a `return` in front so that Perl realizes the opening brace isn't starting a BLOCK. The economy and mnemonic value of using curlies is deemed worth this occasional extra hassle.

For example, if you wanted a function to make a new hash and return a reference to it, you have these options:

```
sub hashem {          { @_ } }    # silently wrong
sub hashem {          +{ @_ } }    # ok
sub hashem { return { @_ } }    # ok
```

On the other hand, if you want the other meaning, you can do this:

```
sub showem {          { @_ } }    # ambiguous (currently ok, but may change)
sub showem {          {; @_ } }    # ok
sub showem { { return @_ } }    # ok
```

The leading `+{` and `{;` always serve to disambiguate the expression to mean either the HASH reference, or the BLOCK.

4. A reference to an anonymous subroutine can be created by using `sub` without a subname:

```
$coderef = sub { print "Boink!\n" };
```

Note the semicolon. Except for the code inside not being immediately executed, a `sub {}` is not so much a declaration as it is an operator, like `do{}` or `eval{}`. (However, no matter how many times you execute that particular line (unless you're in an `eval("...")`), `$coderef` will still have a reference to the *same* anonymous subroutine.)

Anonymous subroutines act as closures with respect to `my()` variables, that is, variables lexically visible within the current scope. Closure is a notion out of the Lisp world that says if you define an anonymous function in a particular lexical context, it pretends to run in that context even when it's called outside the context.

In human terms, it's a funny way of passing arguments to a subroutine when you define it as well as when you call it. It's useful for setting up little bits of code to run later, such as callbacks. You can even do object-oriented stuff with it, though Perl already provides a different mechanism to do that—see [perlobj](#).

You might also think of closure as a way to write a subroutine template without using `eval()`. Here's a small example of how closures work:

```
sub newprint {
    my $x = shift;
    return sub { my $y = shift; print "$x, $y!\n"; };
}
$h = newprint("Howdy");
$g = newprint("Greetings");

# Time passes...

&$h("world");
&$g("earthlings");
```

This prints

```
Howdy, world!
Greetings, earthlings!
```

Note particularly that `$x` continues to refer to the value passed into `newprint()` *despite* "my `$x`" having gone out of scope by the time the anonymous subroutine runs. That's what a closure is all about.

This applies only to lexical variables, by the way. Dynamic variables continue to work as they have always worked. Closure is not something that most Perl programmers need trouble themselves about to begin with.

5. References are often returned by special subroutines called constructors. Perl objects are just references to a special type of object that happens to know which package it's associated with. Constructors are just special subroutines that know how to create that association. They do so by starting with an ordinary reference, and it remains an ordinary reference even while it's also being an object. Constructors are often named `new()` and called indirectly:

```
$objref = new Doggie (Tail => 'short', Ears => 'long');
```

But don't have to be:

```
$objref = Doggie->new(Tail => 'short', Ears => 'long');

use Term::Cap;
$terminal = Term::Cap->Tgetent( { OSPEED => 9600 });

use Tk;
$main = MainWindow->new();
$menu = $main->Frame(-relief           => "raised",
                    -borderwidth      => 2)
```

6. References of the appropriate type can spring into existence if you dereference them in a context that assumes they exist. Because we haven't talked about dereferencing yet, we can't show you any examples yet.
7. A reference can be created by using a special syntax, lovingly known as the `*foo{THING}` syntax. `*foo{THING}` returns a reference to the `THING` slot in `*foo` (which is the symbol table entry which holds everything known as `foo`).

```
$scalarref = *foo{SCALAR};
$arrayref  = *ARGV{ARRAY};
```

```

$hashref    = *ENV{HASH};
$coderef    = *handler{CODE};
$ioref      = *STDIN{IO};
$globref    = *foo{GLOB};

```

All of these are self-explanatory except for `*foo{IO}`. It returns the IO handle, used for file handles (*open*), sockets (*socket* and *socketpair*), and directory handles (*opendir*). For compatibility with previous versions of Perl, `*foo{FILEHANDLE}` is a synonym for `*foo{IO}`, though it is deprecated as of 5.8.0. If deprecation warnings are in effect, it will warn of its use.

`*foo{THING}` returns undef if that particular THING hasn't been used yet, except in the case of scalars. `*foo{SCALAR}` returns a reference to an anonymous scalar if `$foo` hasn't been used yet. This might change in a future release.

`*foo{IO}` is an alternative to the `*HANDLE` mechanism given in *Typeglobs and Filehandles in perldata* for passing filehandles into or out of subroutines, or storing into larger data structures. Its disadvantage is that it won't create a new filehandle for you. Its advantage is that you have less risk of clobbering more than you want to with a typeglob assignment. (It still conflates file and directory handles, though.) However, if you assign the incoming value to a scalar instead of a typeglob as we do in the examples below, there's no risk of that happening.

```

splutter(*STDOUT);           # pass the whole glob
splutter(*STDOUT{IO});       # pass both file and dir handles

sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(*STDIN);       # pass the whole glob
$rec = get_rec(*STDIN{IO});   # pass both file and dir handles

sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}

```

Using References

That's it for creating references. By now you're probably dying to know how to use references to get back to your long-lost data. There are several basic methods.

1. Anywhere you'd put an identifier (or chain of identifiers) as part of a variable or subroutine name, you can replace the identifier with a simple scalar variable containing a reference of the correct type:

```

$bar = $$scalarref;
push(@$arrayref, $filename);
$$arrayref[0] = "January";
$$hashref{"KEY"} = "VALUE";
&$coderef(1,2,3);
print $globref "output\n";

```

It's important to understand that we are specifically *not* dereferencing `$arrayref[0]` or `$hashref{"KEY"}` there. The dereference of the scalar variable happens *before* it does any key lookups. Anything more complicated than a simple scalar variable must use methods 2 or 3 below. However, a "simple scalar" includes an identifier that itself uses method 1 recursively. Therefore, the following prints "howdy".

```

$refrefref = \\\"howdy";
print $$$refrefref;

```

2. Anywhere you'd put an identifier (or chain of identifiers) as part of a variable or subroutine name, you can replace the identifier with a BLOCK returning a reference of the correct type. In other words, the previous examples could be written like this:

```
$bar = ${$scalarref};
push(@{$arrayref}, $filename);
${$arrayref}[0] = "January";
${$hashref}{"KEY"} = "VALUE";
&{$coderef}(1,2,3);
$globref->print("output\n"); # iff IO::Handle is loaded
```

Admittedly, it's a little silly to use the curlies in this case, but the BLOCK can contain any arbitrary expression, in particular, subscripted expressions:

```
&{ $dispatch{$index} }(1,2,3); # call correct routine
```

Because of being able to omit the curlies for the simple case of `$$x`, people often make the mistake of viewing the dereferencing symbols as proper operators, and wonder about their precedence. If they were, though, you could use parentheses instead of braces. That's not the case. Consider the difference below; case 0 is a short-hand version of case 1, *not* case 2:

```
$$hashref{"KEY"} = "VALUE"; # CASE 0
${$hashref}{"KEY"} = "VALUE"; # CASE 1
${$hashref{"KEY"}} = "VALUE"; # CASE 2
${$hashref->{"KEY"}} = "VALUE"; # CASE 3
```

Case 2 is also deceptive in that you're accessing a variable called `%hashref`, not dereferencing through `$hashref` to the hash it's presumably referencing. That would be case 3.

3. Subroutine calls and lookups of individual array elements arise often enough that it gets cumbersome to use method 2. As a form of syntactic sugar, the examples for method 2 may be written:

```
$arrayref->[0] = "January"; # Array element
$hashref->{"KEY"} = "VALUE"; # Hash element
$coderef->(1,2,3); # Subroutine call
```

The left side of the arrow can be any expression returning a reference, including a previous dereference. Note that `$array[$x]` is *not* the same thing as `< $array-[$x]` here:

```
$array[$x]->{"foo"}->[0] = "January";
```

This is one of the cases we mentioned earlier in which references could spring into existence when in an lvalue context. Before this statement, `$array[$x]` may have been undefined. If so, it's automatically defined with a hash reference so that we can look up `{"foo"}` in it. Likewise `< $array[$x]-{"foo"}` will automatically get defined with an array reference so that we can look up `[0]` in it. This process is called *autovivification*.

One more thing here. The arrow is optional *between* brackets subscripts, so you can shrink the above down to

```
$array[$x>{"foo"}[0] = "January";
```

Which, in the degenerate case of using only ordinary arrays, gives you multidimensional arrays just like C's:

```
$score[$x][$y][$z] += 42;
```

Well, okay, not entirely like C's arrays, actually. C doesn't know how to grow its arrays on demand. Perl does.

4. If a reference happens to be a reference to an object, then there are probably methods to access the things referred to, and you should probably stick to those methods unless you're in the class package that defines the object's methods. In other words, be nice, and don't violate the object's encapsulation without a very good reason. Perl does not enforce encapsulation. We are not totalitarians here. We do expect some basic civility though.

Using a string or number as a reference produces a symbolic reference, as explained above. Using a reference as a number produces an integer representing its storage location in memory. The only useful thing to be done with this is to compare two references numerically to see whether they refer to the same location.

```
if ($ref1 == $ref2) { # cheap numeric compare of references
    print "refs 1 and 2 refer to the same thing\n";
}
```

Using a reference as a string produces both its referent's type, including any package blessing as described in [perlobj](#), as well as the numeric address expressed in hex. The `ref()` operator returns just the type of thing the reference is pointing to, without the address. See [ref](#) for details and examples of its use.

The `bless()` operator may be used to associate the object a reference points to with a package functioning as an object class. See [perlobj](#).

A typeglob may be dereferenced the same way a reference can, because the dereference syntax always indicates the type of reference desired. So `$_{*foo}` and `$_{\$foo}` both indicate the same scalar variable.

Here's a trick for interpolating a subroutine call into a string:

```
print "My sub returned @{[mysub(1,2,3)]} that time.\n";
```

The way it works is that when the `@{...}` is seen in the double-quoted string, it's evaluated as a block. The block creates a reference to an anonymous array containing the results of the call to `mysub(1,2,3)`. So the whole block returns a reference to an array, which is then dereferenced by `@{...}` and stuck into the double-quoted string. This chicanery is also useful for arbitrary expressions:

```
print "That yields @{[$n + 5]} widgets\n";
```

Symbolic references

We said that references spring into existence as necessary if they are undefined, but we didn't say what happens if a value used as a reference is already defined, but *isn't* a hard reference. If you use it as a reference, it'll be treated as a symbolic reference. That is, the value of the scalar is taken to be the *name* of a variable, rather than a direct link to a (possibly) anonymous value.

People frequently expect it to work like this. So it does.

```
$name = "foo";
$$name = 1;           # Sets $foo
${$name} = 2;         # Sets $foo
${$name x 2} = 3;     # Sets $foofoo
$name->[0] = 4;        # Sets $foo[0]
@$name = ();          # Clears @foo
&$name();             # Calls &foo() (as in Perl 4)
$pack = "THAT";
${"${pack}::$name"} = 5; # Sets $THAT::foo without eval
```

This is powerful, and slightly dangerous, in that it's possible to intend (with the utmost sincerity) to use a hard reference, and accidentally use a symbolic reference instead. To protect against that, you can say

```
use strict 'refs';
```

and then only hard references will be allowed for the rest of the enclosing block. An inner block may countermand that with


```
no strict 'refs';
```

Only package variables (globals, even if localized) are visible to symbolic references. Lexical variables (declared with `my()`) aren't in a symbol table, and thus are invisible to this mechanism. For example:

```
local $value = 10;
$ref = "value";
{
    my $value = 20;
    print $$ref;
}
```

This will still print 10, not 20. Remember that `local()` affects package variables, which are all "global" to the package.

Not-so-symbolic references

A new feature contributing to readability in perl version 5.001 is that the brackets around a symbolic reference behave more like quotes, just as they always have within a string. That is,

```
$push = "pop on ";
print "${push}over";
```

has always meant to print "pop on over", even though `push` is a reserved word. This has been generalized to work the same outside of quotes, so that

```
print ${push} . "over";
```

and even

```
print ${ push } . "over";
```

will have the same effect. (This would have been a syntax error in Perl 5.000, though Perl 4 allowed it in the spaceless form.) This construct is *not* considered to be a symbolic reference when you're using strict refs:

```
use strict 'refs';
${ bareword };      # Okay, means $bareword.
${ "bareword" };    # Error, symbolic reference.
```

Similarly, because of all the subscripting that is done using single words, we've applied the same rule to any bareword that is used for subscripting a hash. So now, instead of writing

```
$array{ "aaa" }{ "bbb" }{ "ccc" }
```

you can write just

```
$array{ aaa }{ bbb }{ ccc }
```

and not worry about whether the subscripts are reserved words. In the rare event that you do wish to do something like

```
$array{ shift }
```

you can force interpretation as a reserved word by adding anything that makes it more than a bareword:

```
$array{ shift() }
$array{ +shift }
$array{ shift @_ }
```

The `use warnings` pragma or the `-w` switch will warn you if it interprets a reserved word as a string. But it will no longer warn you about using lowercase words, because the string is effectively quoted.

Pseudo-hashes: Using an array as a hash

WARNING: This section describes an experimental feature. Details may change without notice in future versions.

NOTE: The current user-visible implementation of pseudo-hashes (the weird use of the first array element) is deprecated starting from Perl 5.8.0 and will be removed in Perl 5.10.0, and the feature will be implemented differently. Not only is the current interface rather ugly, but the current implementation slows down normal array and hash use quite noticeably. The ‘fields’ pragma interface will remain available.

Beginning with release 5.005 of Perl, you may use an array reference in some contexts that would normally require a hash reference. This allows you to access array elements using symbolic names, as if they were fields in a structure.

For this to work, the array must contain extra information. The first element of the array has to be a hash reference that maps field names to array indices. Here is an example:

```
$struct = [{foo => 1, bar => 2}, "FOO", "BAR"];

$struct->{foo}; # same as $struct->[1], i.e. "FOO"
$struct->{bar}; # same as $struct->[2], i.e. "BAR"

keys %$struct; # will return ("foo", "bar") in some order
values %$struct; # will return ("FOO", "BAR") in same some order

while (my($k,$v) = each %$struct) {
    print "$k => $v\n";
}
```

Perl will raise an exception if you try to access nonexistent fields. To avoid inconsistencies, always use the `fields::phash()` function provided by the `fields` pragma.

```
use fields;
$pseudohash = fields::phash(foo => "FOO", bar => "BAR");
```

For better performance, Perl can also do the translation from field names to array indices at compile time for typed object references. See [fields](#).

There are two ways to check for the existence of a key in a pseudo-hash. The first is to use `exists()`. This checks to see if the given field has ever been set. It acts this way to match the behavior of a regular hash. For instance:

```
use fields;
$phash = fields::phash([qw(foo bar pants)], ['FOO']);
$phash->{pants} = undef;

print exists $phash->{foo}; # true, 'foo' was set in the declaration
print exists $phash->{bar}; # false, 'bar' has not been used.
print exists $phash->{pants}; # true, your 'pants' have been touched
```

The second is to use `exists()` on the hash reference sitting in the first array element. This checks to see if the given key is a valid field in the pseudo-hash.

```
print exists $phash->[0]{bar}; # true, 'bar' is a valid field
print exists $phash->[0]{shoes}; # false, 'shoes' can't be used
```

`delete()` on a pseudo-hash element only deletes the value corresponding to the key, not the key itself. To delete the key, you'll have to explicitly delete it from the first hash element.

```
print delete $phash->{foo}; # prints $phash->[1], "FOO"
print exists $phash->{foo}; # false
print exists $phash->[0]{foo}; # true, key still exists
print delete $phash->[0]{foo}; # now key is gone
print $phash->{foo}; # runtime exception
```

Function Templates

As explained above, a closure is an anonymous function with access to the lexical variables visible when that function was compiled. It retains access to those variables even though it doesn't get run until later, such as in a signal handler or a Tk callback.

Using a closure as a function template allows us to generate many functions that act similarly. Suppose you wanted functions named after the colors that generated HTML font changes for the various colors:

```
print "Be ", red("careful"), "with that ", green("light");
```

The `red()` and `green()` functions would be similar. To create these, we'll assign a closure to a typeglob of the name of the function we're trying to build.

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
    no strict 'refs';          # allow symbol table manipulation
    *$name = *{uc $name} = sub { "<FONT COLOR=' $name' >@_</FONT>" };
}
```

Now all those different functions appear to exist independently. You can call `red()`, `RED()`, `blue()`, `BLUE()`, `green()`, etc. This technique saves on both compile time and memory use, and is less error-prone as well, since syntax checks happen at compile time. It's critical that any variables in the anonymous subroutine be lexicals in order to create a proper closure. That's the reasons for the `my` on the loop iteration variable.

This is one of the only places where giving a prototype to a closure makes much sense. If you wanted to impose scalar context on the arguments of these functions (probably not a wise idea for this particular example), you could have written it this way instead:

```
*$name = sub ($) { "<FONT COLOR=' $name' >$_[0]</FONT>" };
```

However, since prototype checking happens at compile time, the assignment above happens too late to be of much use. You could address this by putting the whole loop of assignments within a `BEGIN` block, forcing it to occur during compilation.

Access to lexicals that change over type—like those in the `for` loop above—only works with closures, not general subroutines. In the general case, then, named subroutines do not nest properly, although anonymous ones do. If you are accustomed to using nested subroutines in other programming languages with their own private variables, you'll have to work at it a bit in Perl. The intuitive coding of this type of thing incurs mysterious warnings about “will not stay shared”. For example, this won't work:

```
sub outer {
    my $x = $_[0] + 35;
    sub inner { return $x * 19 }    # WRONG
    return $x + inner();
}
```

A work-around is the following:

```
sub outer {
    my $x = $_[0] + 35;
    local *inner = sub { return $x * 19 };
    return $x + inner();
}
```

Now `inner()` can only be called from within `outer()`, because of the temporary assignments of the closure (anonymous subroutine). But when it does, it has normal access to the lexical variable `$x` from the scope of `outer()`.

This has the interesting effect of creating a function local to another function, something not normally supported in Perl.

WARNING

You may not (usefully) use a reference as the key to a hash. It will be converted into a string:

```
$x{ \ $a } = $a;
```

If you try to dereference the key, it won't do a hard dereference, and you won't accomplish what you're attempting. You might want to do something more like

```
$r = \@a;  
$x{ $r } = $r;
```

And then at least you can use the `values()`, which will be real refs, instead of the `keys()`, which won't.

The standard `Tie::RefHash` module provides a convenient workaround to this.

SEE ALSO

Besides the obvious documents, source code can be instructive. Some pathological examples of the use of references can be found in the *t/op/ref.t* regression test in the Perl source directory.

See also [perldsc](#) and [perllool](#) for how to use references to create complex data structures, and [perltoot](#), [perlobj](#), and [perlbot](#) for how to use them to create objects.

NAME

perlreftut – Mark's very short tutorial about references

DESCRIPTION

One of the most important new features in Perl 5 was the capability to manage complicated data structures like multidimensional arrays and nested hashes. To enable these, Perl 5 introduced a feature called 'references', and using references is the key to managing complicated, structured data in Perl. Unfortunately, there's a lot of funny syntax to learn, and the main manual page can be hard to follow. The manual is quite complete, and sometimes people find that a problem, because it can be hard to tell what is important and what isn't.

Fortunately, you only need to know 10% of what's in the main page to get 90% of the benefit. This page will show you that 10%.

Who Needs Complicated Data Structures?

One problem that came up all the time in Perl 4 was how to represent a hash whose values were lists. Perl 4 had hashes, of course, but the values had to be scalars; they couldn't be lists.

Why would you want a hash of lists? Let's take a simple example: You have a file of city and country names, like this:

```
Chicago, USA
Frankfurt, Germany
Berlin, Germany
Washington, USA
Helsinki, Finland
New York, USA
```

and you want to produce an output like this, with each country mentioned once, and then an alphabetical list of the cities in that country:

```
Finland: Helsinki.
Germany: Berlin, Frankfurt.
USA: Chicago, New York, Washington.
```

The natural way to do this is to have a hash whose keys are country names. Associated with each country name key is a list of the cities in that country. Each time you read a line of input, split it into a country and a city, look up the list of cities already known to be in that country, and append the new city to the list. When you're done reading the input, iterate over the hash as usual, sorting each list of cities before you print it out.

If hash values can't be lists, you lose. In Perl 4, hash values can't be lists; they can only be strings. You lose. You'd probably have to combine all the cities into a single string somehow, and then when time came to write the output, you'd have to break the string into a list, sort the list, and turn it back into a string. This is messy and error-prone. And it's frustrating, because Perl already has perfectly good lists that would solve the problem if only you could use them.

The Solution

By the time Perl 5 rolled around, we were already stuck with this design: Hash values must be scalars. The solution to this is references.

A reference is a scalar value that *refers to* an entire array or an entire hash (or to just about anything else). Names are one kind of reference that you're already familiar with. Think of the President of the United States: a messy, inconvenient bag of blood and bones. But to talk about him, or to represent him in a computer program, all you need is the easy, convenient scalar string "George Bush".

References in Perl are like names for arrays and hashes. They're Perl's private, internal names, so you can be sure they're unambiguous. Unlike "George Bush", a reference only refers to one thing, and you always know what it refers to. If you have a reference to an array, you can recover the entire array from it. If you have a reference to a hash, you can recover the entire hash. But the reference is still an easy, compact scalar

value.

You can't have a hash whose values are arrays; hash values can only be scalars. We're stuck with that. But a single reference can refer to an entire array, and references are scalars, so you can have a hash of references to arrays, and it'll act a lot like a hash of arrays, and it'll be just as useful as a hash of arrays.

We'll come back to this city-country problem later, after we've seen some syntax for managing references.

Syntax

There are just two ways to make a reference, and just two ways to use it once you have it.

Making References

Make Rule 1

If you put a `\` in front of a variable, you get a reference to that variable.

```
$aref = \@array;      # $aref now holds a reference to @array
$href = \%hash;      # $href now holds a reference to %hash
```

Once the reference is stored in a variable like `$aref` or `$href`, you can copy it or store it just the same as any other scalar value:

```
$xy = $aref;          # $xy now holds a reference to @array
$p[3] = $href;        # $p[3] now holds a reference to %hash
$z = $p[3];           # $z now holds a reference to %hash
```

These examples show how to make references to variables with names. Sometimes you want to make an array or a hash that doesn't have a name. This is analogous to the way you like to be able to use the string `"\n"` or the number 80 without having to store it in a named variable first.

Make Rule 2

`[ITEMS]` makes a new, anonymous array, and returns a reference to that array. `{ ITEMS }` makes a new, anonymous hash, and returns a reference to that hash.

```
$aref = [ 1, "foo", undef, 13 ];
# $aref now holds a reference to an array

$href = { APR => 4, AUG => 8 };
# $href now holds a reference to a hash
```

The references you get from rule 2 are the same kind of references that you get from rule 1:

```
# This:
$aref = [ 1, 2, 3 ];

# Does the same as this:
@array = (1, 2, 3);
$aref = \@array;
```

The first line is an abbreviation for the following two lines, except that it doesn't create the superfluous array variable `@array`.

Using References

What can you do with a reference once you have it? It's a scalar value, and we've seen that you can store it as a scalar and get it back again just like any scalar. There are just two more ways to use it:

Use Rule 1

If `$aref` contains a reference to an array, then you can put `{ $aref }` anywhere you would normally put the name of an array. For example, `@{ $aref }` instead of `@array`.

Here are some examples of that:

Arrays:

<code>@a</code>	<code>@{\$aref}</code>	An array
<code>reverse @a</code>	<code>reverse @{\$aref}</code>	Reverse the array
<code>\$a[3]</code>	<code>\${\$aref}[3]</code>	An element of the array
<code>\$a[3] = 17;</code>	<code>\${\$aref}[3] = 17</code>	Assigning an element

On each line are two expressions that do the same thing. The left-hand versions operate on the array `@a`, and the right-hand versions operate on the array that is referred to by `$aref`, but once they find the array they're operating on, they do the same things to the arrays.

Using a hash reference is *exactly* the same:

<code>%h</code>	<code>%{\$href}</code>	A hash
<code>keys %h</code>	<code>keys %{\$href}</code>	Get the keys from the hash
<code>\$h{'red'}</code>	<code>\${\$href}{'red'}</code>	An element of the hash
<code>\$h{'red'} = 17</code>	<code>\${\$href}{'red'} = 17</code>	Assigning an element

Use Rule 2

`${$aref}[3]` is too hard to read, so you can write `< $aref-[3]` instead.

`${$href}{red}` is too hard to read, so you can write `< $href-{red}` instead.

Most often, when you have an array or a hash, you want to get or set a single element from it. `${$aref}[3]` and `${$href}{'red'}` have too much punctuation, and Perl lets you abbreviate.

If `$aref` holds a reference to an array, then `< $aref-[3]` is the fourth element of the array. Don't confuse this with `$aref[3]`, which is the fourth element of a totally different array, one deceptively named `@aref`. `$aref` and `@aref` are unrelated the same way that `$item` and `@item` are.

Similarly, `< $href-{ 'red' }` is part of the hash referred to by the scalar variable `$href`, perhaps even one with no name. `$href{'red'}` is part of the deceptively named `%href` hash. It's easy to forget to leave out the `< -`, and if you do, you'll get bizarre results when your program gets array and hash elements out of totally unexpected hashes and arrays that weren't the ones you wanted to use.

An Example

Let's see a quick example of how all this is useful.

First, remember that `[1, 2, 3]` makes an anonymous array containing `(1, 2, 3)`, and gives you a reference to that array.

Now think about

```
@a = ( [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
      );
```

`@a` is an array with three elements, and each one is a reference to another array.

`$a[1]` is one of these references. It refers to an array, the array containing `(4, 5, 6)`, and because it is a reference to an array, **USE RULE 2** says that we can write `< $a[1]-[2]` to get the third element from that array. `< $a[1]-[2]` is the 6. Similarly, `< $a[0]-[1]` is the 2. What we have here is like a two-dimensional array; you can write `< $a[ROW]-[COLUMN]` to get or set the element in any row and any column of the array.

The notation still looks a little cumbersome, so there's one more abbreviation:

Arrow Rule

In between two **subscripts**, the arrow is optional.

Instead of `< $a[1]-[2]`, we can write `$a[1][2]`; it means the same thing. Instead of `< $a[0]-[1]`, we can write `$a[0][1]`; it means the same thing.

Now it really looks like two-dimensional arrays!

You can see why the arrows are important. Without them, we would have had to write `#{ $a[1] } [2]` instead of `$a[1] [2]`. For three-dimensional arrays, they let us write `$x[2] [3] [5]` instead of the unreadable `#{ $x[2] } [3] } [5]`.

Solution

Here's the answer to the problem I posed earlier, of reformatting a file of city and country names.

```

1  while (<>) {
2      chomp;
3      my ($city, $country) = split /, /;
4      push @{$table{$country}}, $city;
5  }
6
7  foreach $country (sort keys %table) {
8      print "$country: ";
9      my @cities = @{$table{$country}};
10     print join ' ', ' ', sort @cities;
11     print ".\n";
12 }
```

The program has two pieces: Lines 1—5 read the input and build a data structure, and lines 7—12 analyze the data and print out the report.

In the first part, line 4 is the important one. We're going to have a hash, `%table`, whose keys are country names, and whose values are (references to) arrays of city names. After acquiring a city and country name, the program looks up `$table{$country}`, which holds (a reference to) the list of cities seen in that country so far. Line 4 is totally analogous to

```
push @array, $city;
```

except that the name `array` has been replaced by the reference `{ $table{$country} }`. The `push` adds a city name to the end of the referred-to array.

In the second part, line 9 is the important one. Again, `$table{$country}` is (a reference to) the list of cities in the country, so we can recover the original list, and copy it into the array `@cities`, by using `@{ $table{$country} }`. Line 9 is totally analogous to

```
@cities = @array;
```

except that the name `array` has been replaced by the reference `{ $table{$country} }`. The `@` tells Perl to get the entire array.

The rest of the program is just familiar uses of `chomp`, `split`, `sort`, `print`, and doesn't involve references at all.

There's one fine point I skipped. Suppose the program has just read the first line in its input that happens to mention Greece. Control is at line 4, `$country` is 'Greece', and `$city` is 'Athens'. Since this is the first city in Greece, `$table{$country}` is undefined—in fact there isn't an 'Greece' key in `%table` at all. What does line 4 do here?

```
4      push @{$table{$country}}, $city;
```

This is Perl, so it does the exact right thing. It sees that you want to push `Athens` onto an array that doesn't exist, so it helpfully makes a new, empty, anonymous array for you, installs it in the table, and then pushes `Athens` onto it. This is called 'autovivification'.

The Rest

I promised to give you 90% of the benefit with 10% of the details, and that means I left out 90% of the details. Now that you have an overview of the important parts, it should be easier to read the [perlref](#) manual page, which discusses 100% of the details.

Some of the highlights of [perlref](#):

- You can make references to anything, including scalars, functions, and other references.
- In **USE RULE 1**, you can omit the curly brackets whenever the thing inside them is an atomic scalar variable like `$aref`. For example, `@$aref` is the same as `@{$aref}`, and `$$aref[1]` is the same as `${$aref}[1]`. If you're just starting out, you may want to adopt the habit of always including the curly brackets.
- To see if a variable contains a reference, use the `'ref'` function. It returns true if its argument is a reference. Actually it's a little better than that: It returns `HASH` for hash references and `ARRAY` for array references.
- If you try to use a reference like a string, you get strings like

```
ARRAY(0x80f5dec)    or    HASH(0x826afc0)
```

If you ever see a string that looks like this, you'll know you printed out a reference by mistake.

A side effect of this representation is that you can use `eq` to see if two references refer to the same thing. (But you should usually use `==` instead because it's much faster.)

- You can use a string as if it were a reference. If you use the string `"foo"` as an array reference, it's taken to be a reference to the array `@foo`. This is called a *soft reference* or *symbolic reference*.

You might prefer to go on to [perllob](#) instead of [perlref](#); it discusses lists of lists and multidimensional arrays in detail. After that, you should move on to [perldsc](#); it's a Data Structure Cookbook that shows recipes for using and printing out arrays of hashes, hashes of arrays, and other kinds of data.

Summary

Everyone needs compound data structures, and in Perl the way you get them is with references. There are four important rules for managing references: Two for making references and two for using them. Once you know these rules you can do most of the important things you need to do with references.

Credits

Author: Mark-Jason Dominus, Plover Systems (mjd-perl-ref+@plover.com)

This article originally appeared in *The Perl Journal* (<http://www.tpj.com/>) volume 3, #2. Reprinted with permission.

The original title was *Understand References Today*.

Distribution Conditions

Copyright 1998 The Perl Journal.

When included as part of the Standard Version of Perl, or as part of its complete documentation whether printed or otherwise, this work may be distributed only under the terms of Perl's Artistic License. Any distribution of this file or derivatives thereof outside of that package require that special arrangements be made with copyright holder.

Irrespective of its distribution, all code examples in these files are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

NAME

perlrequick – Perl regular expressions quick start

DESCRIPTION

This page covers the very basics of understanding, creating and using regular expressions (‘regexes’) in Perl.

The Guide**Simple word matching**

The simplest regex is simply a word, or more generally, a string of characters. A regex consisting of a word matches any string that contains that word:

```
"Hello World" =~ /World/; # matches
```

In this statement, `World` is a regex and the `//` enclosing `/World/` tells perl to search a string for a match. The operator `=~` associates the string with the regex match and produces a true value if the regex matched, or false if the regex did not match. In our case, `World` matches the second word in `"Hello World"`, so the expression is true. This idea has several variations.

Expressions like this are useful in conditionals:

```
print "It matches\n" if "Hello World" =~ /World/;
```

The sense of the match can be reversed by using `!~` operator:

```
print "It doesn't match\n" if "Hello World" !~ /World/;
```

The literal string in the regex can be replaced by a variable:

```
$greeting = "World";
print "It matches\n" if "Hello World" =~ /$greeting/;
```

If you're matching against `$_`, the `$_ =~` part can be omitted:

```
$_ = "Hello World";
print "It matches\n" if /World/;
```

Finally, the `//` default delimiters for a match can be changed to arbitrary delimiters by putting an `'m'` out front:

```
"Hello World" =~ m!World!; # matches, delimited by '!'
"Hello World" =~ m{World}; # matches, note the matching '{} '
"/usr/bin/perl" =~ m"/perl"; # matches after '/usr/bin',
                             # '/' becomes an ordinary char
```

Regexes must match a part of the string *exactly* in order for the statement to be true:

```
"Hello World" =~ /world/; # doesn't match, case sensitive
"Hello World" =~ /o W/;   # matches, ' ' is an ordinary char
"Hello World" =~ /World /; # doesn't match, no ' ' at end
```

perl will always match at the earliest possible point in the string:

```
"Hello World" =~ /o/;      # matches 'o' in 'Hello'
"That hat is red" =~ /hat/; # matches 'hat' in 'That'
```

Not all characters can be used ‘as is’ in a match. Some characters, called **metacharacters**, are reserved for use in regex notation. The metacharacters are

```
{ } [ ] ( ) ^ $ . | * + ? \
```

A metacharacter can be matched by putting a backslash before it:

```
"2+2=4" =~ /2+2/;      # doesn't match, + is a metacharacter
"2+2=4" =~ /2\+2/;     # matches, \+ is treated like an ordinary +
'C:\WIN32' =~ /C:\\WIN/; # matches
"/usr/bin/perl" =~ /\usr\local\bin\perl/; # matches
```

In the last regex, the forward slash '/' is also backslashed, because it is used to delimit the regex.

Non-printable ASCII characters are represented by **escape sequences**. Common examples are \t for a tab, \n for a newline, and \r for a carriage return. Arbitrary bytes are represented by octal escape sequences, e.g., \033, or hexadecimal escape sequences, e.g., \x1B:

```
"1000\t2000" =~ m(0\t2) # matches
"cat" =~ /\143\x61\x74/ # matches, but a weird way to spell cat
```

Regexes are treated mostly as double quoted strings, so variable substitution works:

```
$foo = 'house';
'cathouse' =~ /cat$foo/; # matches
'housecat' =~ /${foo}cat/; # matches
```

With all of the regexes above, if the regex matched anywhere in the string, it was considered a match. To specify *where* it should match, we would use the **anchor** metacharacters ^ and \$. The anchor ^ means match at the beginning of the string and the anchor \$ means match at the end of the string, or before a newline at the end of the string. Some examples:

```
"housekeeper" =~ /keeper/; # matches
"housekeeper" =~ /^keeper/; # doesn't match
"housekeeper" =~ /keeper$/; # matches
"housekeeper\n" =~ /keeper$/; # matches
"housekeeper" =~ /^housekeeper$/; # matches
```

Using character classes

A **character class** allows a set of possible characters, rather than just a single character, to match at a particular point in a regex. Character classes are denoted by brackets [...], with the set of characters to be possibly matched inside. Here are some examples:

```
/cat/; # matches 'cat'
/[bcr]at/; # matches 'bat', 'cat', or 'rat'
"abc" =~ /[cab]/; # matches 'a'
```

In the last statement, even though 'c' is the first character in the class, the earliest point at which the regex can match is 'a'.

```
/[yY][eE][sS]/; # match 'yes' in a case-insensitive way
                # 'yes', 'Yes', 'YES', etc.
/yes/i; # also match 'yes' in a case-insensitive way
```

The last example shows a match with an 'i' **modifier**, which makes the match case-insensitive.

Character classes also have ordinary and special characters, but the sets of ordinary and special characters inside a character class are different than those outside a character class. The special characters for a character class are -] ^ \$ and are matched using an escape:

```
/[\\c]def/; # matches 'def' or 'cdef'
$x = 'bcr';
/[$x]at/; # matches 'bat', 'cat', or 'rat'
/[\\$x]at/; # matches '$at' or 'xat'
/[\\$x]at/; # matches '\\at', 'bat', 'cat', or 'rat'
```

The special character '-' acts as a range operator within character classes, so that the unwieldy [0123456789] and [abc...xyz] become the svelte [0-9] and [a-z]:

```
/item[0-9]/; # matches 'item0' or ... or 'item9'
/[0-9a-fA-F]/; # matches a hexadecimal digit
```

If `'-'` is the first or last character in a character class, it is treated as an ordinary character.

The special character `^` in the first position of a character class denotes a **negated character class**, which matches any character but those in the brackets. Both `[...]` and `[^...]` must match a character, or the match fails. Then

```
/[^a]at/; # doesn't match 'aat' or 'at', but matches
           # all other 'bat', 'cat', '0at', '%at', etc.
/[^0-9]/; # matches a non-numeric character
/[a^]at/; # matches 'aat' or '^at'; here '^' is ordinary
```

Perl has several abbreviations for common character classes:

- `\d` is a digit and represents `[0-9]`
- `\s` is a whitespace character and represents `[\t\r\n\f]`
- `\w` is a word character (alphanumeric or `_`) and represents `[0-9a-zA-Z_]`
- `\D` is a negated `\d`; it represents any character but a digit `[^0-9]`
- `\S` is a negated `\s`; it represents any non-whitespace character `[^\s]`
- `\W` is a negated `\w`; it represents any non-word character `[^\w]`
- The period `'.'` matches any character but `"\n"`

The `\d\s\w\D\S\W` abbreviations can be used both inside and outside of character classes. Here are some in use:

```
/\d\d:\d\d:\d\d/; # matches a hh:mm:ss time format
/[\d\s]/;          # matches any digit or whitespace character
/\w\W\w/;          # matches a word char, followed by a
                   # non-word char, followed by a word char
/..rt/;            # matches any two chars, followed by 'rt'
/end\./;           # matches 'end.'
/end[.]/;          # same thing, matches 'end.'
```

The **word anchor** `\b` matches a boundary between a word character and a non-word character `\w\W` or `\W\w`:

```
$x = "Housecat catenates house and cat";
$x =~ /\bcat/; # matches cat in 'catenates'
$x =~ /cat\b/; # matches cat in 'housecat'
$x =~ /\bcat\b/; # matches 'cat' at end of string
```

In the last example, the end of the string is considered a word boundary.

Matching this or that

We can match different character strings with the **alternation** metacharacter `|`. To match `dog` or `cat`, we form the regex `dog|cat`. As before, perl will try to match the regex at the earliest possible point in the string. At each character position, perl will first try to match the first alternative, `dog`. If `dog` doesn't match, perl will then try the next alternative, `cat`. If `cat` doesn't match either, then the match fails and perl moves to the next position in the string. Some examples:

```
"cats and dogs" =~ /cat|dog|bird/; # matches "cat"
"cats and dogs" =~ /dog|cat|bird/; # matches "cat"
```

Even though `dog` is the first alternative in the second regex, `cat` is able to match earlier in the string.

```
"cats" =~ /c|ca|cat|cats/; # matches "c"
```

```
"cats" =~ /cats|cat|ca|c/; # matches "cats"
```

At a given character position, the first alternative that allows the regex match to succeed will be the one that matches. Here, all the alternatives match at the first string position, so the first matches.

Grouping things and hierarchical matching

The **grouping** metacharacters `()` allow a part of a regex to be treated as a single unit. Parts of a regex are grouped by enclosing them in parentheses. The regex `house(cat|keeper)` means match house followed by either cat or keeper. Some more examples are

```
/(a|b)b/;      # matches 'ab' or 'bb'
/^(a|b)c/;     # matches 'ac' at start of string or 'bc' anywhere

/house(cat|)/; # matches either 'housecat' or 'house'
/house(cat(s|)|)/; # matches either 'housecats' or 'housecat' or
                  # 'house'. Note groups can be nested.

"20" =~ /(19|20|)\d\d/; # matches the null alternative '()\d\d',
                        # because '20\d\d' can't match
```

Extracting matches

The grouping metacharacters `()` also allow the extraction of the parts of a string that matched. For each grouping, the part that matched inside goes into the special variables `$1`, `$2`, etc. They can be used just as ordinary variables:

```
# extract hours, minutes, seconds
$time =~ /(\d\d):(\d\d):(\d\d)/; # match hh:mm:ss format
$hours = $1;
$minutes = $2;
$seconds = $3;
```

In list context, a match `/regex/` with groupings will return the list of matched values (`$1, $2, ...`). So we could rewrite it as

```
($hours, $minutes, $seconds) = ($time =~ /(\d\d):(\d\d):(\d\d)/);
```

If the groupings in a regex are nested, `$1` gets the group with the leftmost opening parenthesis, `$2` the next opening parenthesis, etc. For example, here is a complex regex and the matching variables indicated below it:

```
/(ab(cd|ef)((gi)|j))/;
 1  2      34
```

Associated with the matching variables `$1`, `$2`, ... are the **backreferences** `\1`, `\2`, ... Backreferences are matching variables that can be used *inside* a regex:

```
/(\w\w\w)\s\1/; # find sequences like 'the the' in string
```

`$1`, `$2`, ... should only be used outside of a regex, and `\1`, `\2`, ... only inside a regex.

Matching repetitions

The **quantifier** metacharacters `?`, `*`, `+`, and `{ }` allow us to determine the number of repeats of a portion of a regex we consider to be a match. Quantifiers are put immediately after the character, character class, or grouping that we want to specify. They have the following meanings:

- `a?` = match 'a' 1 or 0 times
- `a*` = match 'a' 0 or more times, i.e., any number of times
- `a+` = match 'a' 1 or more times, i.e., at least once

- `a{n,m}` = match at least `n` times, but not more than `m` times.
- `a{n,}` = match at least `n` or more times
- `a{n}` = match exactly `n` times

Here are some examples:

```
[a-z]+\s+\d*/;  # match a lowercase word, at least some space, and
                # any number of digits
/(\w+)\s+\1/;    # match doubled words of arbitrary length
$year =~ /\d{2,4}/; # make sure year is at least 2 but not more
                # than 4 digits
$year =~ /\d{4}|\d{2}/; # better match; throw out 3 digit dates
```

These quantifiers will try to match as much of the string as possible, while still allowing the regex to match. So we have

```
$x = 'the cat in the hat';
$x =~ /^(.*) (at) (.*)$/; # matches,
                        # $1 = 'the cat in the h'
                        # $2 = 'at'
                        # $3 = '' (0 matches)
```

The first quantifier `.*` grabs as much of the string as possible while still having the regex match. The second quantifier `.*` has no string left to it, so it matches 0 times.

More matching

There are a few more things you might want to know about matching operators. In the code

```
$pattern = 'Seuss';
while (<>) {
    print if /$pattern/;
}
```

perl has to re-evaluate `$pattern` each time through the loop. If `$pattern` won't be changing, use the `//o` modifier, to only perform variable substitutions once. If you don't want any substitutions at all, use the special delimiter `m''`:

```
$pattern = 'Seuss';
m'$pattern'; # matches '$pattern', not 'Seuss'
```

The global modifier `//g` allows the matching operator to match within a string as many times as possible. In scalar context, successive matches against a string will have `//g` jump from match to match, keeping track of position in the string as it goes along. You can get or set the position with the `pos()` function. For example,

```
$x = "cat dog house"; # 3 words
while ($x =~ /(\w+)/g) {
    print "Word is $1, ends at position ", pos $x, "\n";
}
```

prints

```
Word is cat, ends at position 3
Word is dog, ends at position 7
Word is house, ends at position 13
```

A failed match or changing the target string resets the position. If you don't want the position reset after failure to match, add the `//c`, as in `/regcxgc`.

In list context, `//g` returns a list of matched groupings, or if there are no groupings, a list of matches to the

whole regex. So

```
@words = ($x =~ /(\w+)/g); # matches,
                        # $word[0] = 'cat'
                        # $word[1] = 'dog'
                        # $word[2] = 'house'
```

Search and replace

Search and replace is performed using `s/regex/replacement/modifiers`. The replacement is a Perl double quoted string that replaces in the string whatever is matched with the regex. The operator `=~` is also used here to associate a string with `s///`. If matching against `$_`, the `$_ =~` can be dropped. If there is a match, `s///` returns the number of substitutions made, otherwise it returns false. Here are a few examples:

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/; # $x contains "Time to feed the hacker!"
$y = "'quoted words'";
$y =~ s/^(.*)'$/\1/; # strip single quotes,
                    # $y contains "quoted words"
```

With the `s///` operator, the matched variables `$1`, `$2`, etc. are immediately available for use in the replacement expression. With the global modifier, `s///g` will search and replace all occurrences of the regex in the string:

```
$x = "I batted 4 for 4";
$x =~ s/4/four/; # $x contains "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g; # $x contains "I batted four for four"
```

The evaluation modifier `s///e` wraps an `eval{...}` around the replacement string and the evaluated result is substituted for the matched substring. Some examples:

```
# reverse all the words in a string
$x = "the cat in the hat";
$x =~ s/(\w+)/reverse $1/ge; # $x contains "eht tac ni eht tah"

# convert percentage to decimal
$x = "A 39% hit rate";
$x =~ s!(\d+)%!$1/100!e; # $x contains "A 0.39 hit rate"
```

The last example shows that `s///` can use other delimiters, such as `s!!!` and `s{ }{ }`, and even `s{ }//`. If single quotes are used `s'''`, then the regex and replacement are treated as single quoted strings.

The split operator

`split /regex/, string` splits `string` into a list of substrings and returns that list. The regex determines the character sequence that `string` is split with respect to. For example, to split a string into words, use

```
$x = "Calvin and Hobbes";
@word = split /\s+/, $x; # $word[0] = 'Calvin'
                        # $word[1] = 'and'
                        # $word[2] = 'Hobbes'
```

To extract a comma-delimited list of numbers, use

```
$x = "1.618,2.718, 3.142";
@const = split /\s*/, $x; # $const[0] = '1.618'
                        # $const[1] = '2.718'
                        # $const[2] = '3.142'
```

If the empty regex `//` is used, the string is split into individual characters. If the regex has groupings, then

list produced contains the matched substrings from the groupings as well:

```
$x = "/usr/bin";
@parts = split m!(/)! , $x;  # $parts[0] = ''
                              # $parts[1] = '/'
                              # $parts[2] = 'usr'
                              # $parts[3] = '/'
                              # $parts[4] = 'bin'
```

Since the first character of `$x` matched the regex, `split` prepended an empty initial element to the list.

BUGS

None.

SEE ALSO

This is just a quick start guide. For a more in-depth tutorial on regexes, see [perlretut](#) and for the reference page, see [perlre](#).

AUTHOR AND COPYRIGHT

Copyright (c) 2000 Mark Kvale All rights reserved.

This document may be distributed under the same terms as Perl itself.

Acknowledgments

The author would like to thank Mark-Jason Dominus, Tom Christiansen, Ilya Zakharevich, Brad Hughes, and Mike Giroux for all their helpful comments.

NAME

perlretut – Perl regular expressions tutorial

DESCRIPTION

This page provides a basic tutorial on understanding, creating and using regular expressions in Perl. It serves as a complement to the reference page on regular expressions [perlre](#). Regular expressions are an integral part of the `m//`, `s///`, `qr//` and `split` operators and so this tutorial also overlaps with [Regex Quote-Like Operators in perlop](#) and [split](#).

Perl is widely renowned for excellence in text processing, and regular expressions are one of the big factors behind this fame. Perl regular expressions display an efficiency and flexibility unknown in most other computer languages. Mastering even the basics of regular expressions will allow you to manipulate text with surprising ease.

What is a regular expression? A regular expression is simply a string that describes a pattern. Patterns are in common use these days; examples are the patterns typed into a search engine to find web pages and the patterns used to list files in a directory, e.g., `ls *.txt` or `dir *.*`. In Perl, the patterns described by regular expressions are used to search strings, extract desired parts of strings, and to do search and replace operations.

Regular expressions have the undeserved reputation of being abstract and difficult to understand. Regular expressions are constructed using simple concepts like conditionals and loops and are no more difficult to understand than the corresponding `if` conditionals and `while` loops in the Perl language itself. In fact, the main challenge in learning regular expressions is just getting used to the terse notation used to express these concepts.

This tutorial flattens the learning curve by discussing regular expression concepts, along with their notation, one at a time and with many examples. The first part of the tutorial will progress from the simplest word searches to the basic regular expression concepts. If you master the first part, you will have all the tools needed to solve about 98% of your needs. The second part of the tutorial is for those comfortable with the basics and hungry for more power tools. It discusses the more advanced regular expression operators and introduces the latest cutting edge innovations in 5.6.0.

A note: to save time, ‘regular expression’ is often abbreviated as `regexp` or `regex`. `Regexp` is a more natural abbreviation than `regex`, but is harder to pronounce. The Perl pod documentation is evenly split on `regexp` vs `regex`; in Perl, there is more than one way to abbreviate it. We’ll use `regexp` in this tutorial.

Part 1: The basics

Simple word matching

The simplest `regexp` is simply a word, or more generally, a string of characters. A `regexp` consisting of a word matches any string that contains that word:

```
"Hello World" =~ /World/; # matches
```

What is this perl statement all about? `"Hello World"` is a simple double quoted string. `World` is the regular expression and the `//` enclosing `/World/` tells perl to search a string for a match. The operator `=~` associates the string with the `regexp` match and produces a true value if the `regexp` matched, or false if the `regexp` did not match. In our case, `World` matches the second word in `"Hello World"`, so the expression is true. Expressions like this are useful in conditionals:

```
if ("Hello World" =~ /World/) {
    print "It matches\n";
}
else {
    print "It doesn't match\n";
}
```

There are useful variations on this theme. The sense of the match can be reversed by using `!~` operator:

```

if ("Hello World" !~ /World/) {
    print "It doesn't match\n";
}
else {
    print "It matches\n";
}

```

The literal string in the regexp can be replaced by a variable:

```

$greeting = "World";
if ("Hello World" =~ /$greeting/) {
    print "It matches\n";
}
else {
    print "It doesn't match\n";
}

```

If you're matching against the special default variable `$_`, the `$_ =~` part can be omitted:

```

$_ = "Hello World";
if (/World/) {
    print "It matches\n";
}
else {
    print "It doesn't match\n";
}

```

And finally, the `//` default delimiters for a match can be changed to arbitrary delimiters by putting an `'m'` out front:

```

"Hello World" =~ m!World!;    # matches, delimited by '!'
"Hello World" =~ m{World};    # matches, note the matching '{}'
"/usr/bin/perl" =~ m"/perl";  # matches after '/usr/bin',
                                # '/' becomes an ordinary char

```

`/World/`, `m!World!`, and `m{World}` all represent the same thing. When, e.g., `"` is used as a delimiter, the forward slash `'/'` becomes an ordinary character and can be used in a regexp without trouble.

Let's consider how different regexps would match "Hello World":

```

"Hello World" =~ /world/;    # doesn't match
"Hello World" =~ /o W/;      # matches
"Hello World" =~ /oW/;       # doesn't match
"Hello World" =~ /World /;   # doesn't match

```

The first regexp `world` doesn't match because regexps are case-sensitive. The second regexp matches because the substring `'o W'` occurs in the string "Hello World". The space character `' '` is treated like any other character in a regexp and is needed to match in this case. The lack of a space character is the reason the third regexp `'oW'` doesn't match. The fourth regexp `'World '` doesn't match because there is a space at the end of the regexp, but not at the end of the string. The lesson here is that regexps must match a part of the string *exactly* in order for the statement to be true.

If a regexp matches in more than one place in the string, perl will always match at the earliest possible point in the string:

```

"Hello World" =~ /o/;        # matches 'o' in 'Hello'
"That hat is red" =~ /hat/;  # matches 'hat' in 'That'

```

With respect to character matching, there are a few more points you need to know about. First of all, not all characters can be used 'as is' in a match. Some characters, called **metacharacters**, are reserved for use in regexp notation. The metacharacters are

```
{ } [ ] ( ) ^ $ . | * + ? \
```

The significance of each of these will be explained in the rest of the tutorial, but for now, it is important only to know that a metacharacter can be matched by putting a backslash before it:

```
"2+2=4" =~ /2+2/;      # doesn't match, + is a metacharacter
"2+2=4" =~ /2\+2/;     # matches, \+ is treated like an ordinary +
"The interval is [0,1). " =~ /[0,1)./      # is a syntax error!
"The interval is [0,1). " =~ /\[0,1\)\/    # matches
"/usr/bin/perl" =~ /\usr\local\bin\perl/;  # matches
```

In the last regexp, the forward slash `'/'` is also backslashed, because it is used to delimit the regexp. This can lead to LTS (leaning toothpick syndrome), however, and it is often more readable to change delimiters.

The backslash character `'\'` is a metacharacter itself and needs to be backslashed:

```
'C:\WIN32' =~ /C:\\WIN/;  # matches
```

In addition to the metacharacters, there are some ASCII characters which don't have printable character equivalents and are instead represented by **escape sequences**. Common examples are `\t` for a tab, `\n` for a newline, `\r` for a carriage return and `\a` for a bell. If your string is better thought of as a sequence of arbitrary bytes, the octal escape sequence, e.g., `\033`, or hexadecimal escape sequence, e.g., `\x1B` may be a more natural representation for your bytes. Here are some examples of escapes:

```
"1000\t2000" =~ m(0\t2)  # matches
"1000\n2000" =~ /0\n20/  # matches
"1000\t2000" =~ /\000\t2/ # doesn't match, "0" ne "\000"
"cat"          =~ /\143\x61\x74/ # matches, but a weird way to spell cat
```

If you've been around Perl a while, all this talk of escape sequences may seem familiar. Similar escape sequences are used in double-quoted strings and in fact the regexps in Perl are mostly treated as double-quoted strings. This means that variables can be used in regexps as well. Just like double-quoted strings, the values of the variables in the regexp will be substituted in before the regexp is evaluated for matching purposes. So we have:

```
$foo = 'house';
'housecat' =~ /$foo/;      # matches
'cathouse' =~ /cat$foo/;   # matches
'housecat' =~ /${foo}cat/; # matches
```

So far, so good. With the knowledge above you can already perform searches with just about any literal string regexp you can dream up. Here is a *very simple* emulation of the Unix `grep` program:

```
% cat > simple_grep
#!/usr/bin/perl
$regexp = shift;
while (<>) {
    print if /$regexp/;
}
^D

% chmod +x simple_grep

% simple_grep abba /usr/dict/words
Babbage
cabbage
cabbages
sabbath
Sabbathize
Sabbathizes
sabbatical
```

```
scabbard
scabbards
```

This program is easy to understand. `#!/usr/bin/perl` is the standard way to invoke a perl program from the shell. `$regex = shift;` saves the first command line argument as the regex to be used, leaving the rest of the command line arguments to be treated as files. `< while (<)` loops over all the lines in all the files. For each line, `print if /$regex/;` prints the line if the regex matches the line. In this line, both `print` and `/ $regex /` use the default variable `$_` implicitly.

With all of the regexps above, if the regex matched anywhere in the string, it was considered a match. Sometimes, however, we'd like to specify *where* in the string the regex should try to match. To do this, we would use the **anchor** metacharacters `^` and `$`. The anchor `^` means match at the beginning of the string and the anchor `$` means match at the end of the string, or before a newline at the end of the string. Here is how they are used:

```
"housekeeper" =~ /keeper/;      # matches
"housekeeper" =~ /^keeper/;     # doesn't match
"housekeeper" =~ /keeper$/;     # matches
"housekeeper\n" =~ /keeper$/;   # matches
```

The second regex doesn't match because `^` constrains `keeper` to match only at the beginning of the string, but "housekeeper" has `keeper` starting in the middle. The third regex does match, since the `$` constrains `keeper` to match only at the end of the string.

When both `^` and `$` are used at the same time, the regex has to match both the beginning and the end of the string, i.e., the regex matches the whole string. Consider

```
"keeper" =~ /^keep$/;          # doesn't match
"keeper" =~ /^keeper$/;        # matches
"" =~ /^$/;                     # ^$ matches an empty string
```

The first regex doesn't match because the string has more to it than `keep`. Since the second regex is exactly the string, it matches. Using both `^` and `$` in a regex forces the complete string to match, so it gives you complete control over which strings match and which don't. Suppose you are looking for a fellow named `bert`, off in a string by himself:

```
"dogbert" =~ /bert/;           # matches, but not what you want
"dilbert" =~ /^bert/;          # doesn't match, but ..
"bertram" =~ /^bert/;          # matches, so still not good enough
"bertram" =~ /^bert$/;         # doesn't match, good
"dilbert" =~ /^bert$/;         # doesn't match, good
"bert" =~ /^bert$/;            # matches, perfect
```

Of course, in the case of a literal string, one could just as easily use the string equivalence `$string eq 'bert'` and it would be more efficient. The `^...$` regex really becomes useful when we add in the more powerful regex tools below.

Using character classes

Although one can already do quite a lot with the literal string regexps above, we've only scratched the surface of regular expression technology. In this and subsequent sections we will introduce regex concepts (and associated metacharacter notations) that will allow a regex to not just represent a single character sequence, but a *whole class* of them.

One such concept is that of a **character class**. A character class allows a set of possible characters, rather than just a single character, to match at a particular point in a regex. Character classes are denoted by brackets `[...]`, with the set of characters to be possibly matched inside. Here are some examples:

```
/cat/;                # matches 'cat'
/[bcr]at/;            # matches 'bat', 'cat', or 'rat'
```

```
/item[0123456789]/; # matches 'item0' or ... or 'item9'
"abc" =~ /[cab]/;   # matches 'a'
```

In the last statement, even though 'c' is the first character in the class, 'a' matches because the first character position in the string is the earliest point at which the regexp can match.

```
/[yY][eE][sS]/;      # match 'yes' in a case-insensitive way
                      # 'yes', 'Yes', 'YES', etc.
```

This regexp displays a common task: perform a case-insensitive match. Perl provides a way of avoiding all those brackets by simply appending an 'i' to the end of the match. Then `/[yY][eE][sS]/;` can be rewritten as `/yes/i;`. The 'i' stands for case-insensitive and is an example of a **modifier** of the matching operation. We will meet other modifiers later in the tutorial.

We saw in the section above that there were ordinary characters, which represented themselves, and special characters, which needed a backslash \ to represent themselves. The same is true in a character class, but the sets of ordinary and special characters inside a character class are different than those outside a character class. The special characters for a character class are `~\^$.]` is special because it denotes the end of a character class. `$` is special because it denotes a scalar variable. `\` is special because it is used in escape sequences, just like above. Here is how the special characters `]$\` are handled:

```
/[\]c]def/; # matches 'def' or 'cdef'
$x = 'bcr';
/[$x]at/;   # matches 'bat', 'cat', or 'rat'
/[\\$x]at/; # matches '$at' or 'xat'
/[\\\$x]at/; # matches '\at', 'bat', 'cat', or 'rat'
```

The last two are a little tricky. In `[\\$x]`, the backslash protects the dollar sign, so the character class has two members `$` and `x`. In `[\\\$x]`, the backslash is protected, so `$x` is treated as a variable and substituted in double quote fashion.

The special character `-` acts as a range operator within character classes, so that a contiguous set of characters can be written as a range. With ranges, the unwieldy `[0123456789]` and `[abc...xyz]` become the svelte `[0-9]` and `[a-z]`. Some examples are

```
/item[0-9]/; # matches 'item0' or ... or 'item9'
/[0-9bx-z]aa/; # matches '0aa', ..., '9aa',
               # 'baa', 'xaa', 'yaa', or 'zaa'
/[0-9a-fA-F]/; # matches a hexadecimal digit
/[0-9a-zA-Z_]/; # matches a "word" character,
               # like those in a perl variable name
```

If `-` is the first or last character in a character class, it is treated as an ordinary character; `[-ab]`, `[ab-]` and `[a\b]` are all equivalent.

The special character `^` in the first position of a character class denotes a **negated character class**, which matches any character but those in the brackets. Both `[...]` and `[^...]` must match a character, or the match fails. Then

```
/[^a]at/; # doesn't match 'aat' or 'at', but matches
           # all other 'bat', 'cat', '0at', '%at', etc.
/[^0-9]/; # matches a non-numeric character
/[a^]at/; # matches 'aat' or '^at'; here '^' is ordinary
```

Now, even `[0-9]` can be a bother to write multiple times, so in the interest of saving keystrokes and making regexps more readable, Perl has several abbreviations for common character classes:

- `\d` is a digit and represents `[0-9]`

- `\s` is a whitespace character and represents `[\t\r\n\f]`
- `\w` is a word character (alphanumeric or `_`) and represents `[0-9a-zA-Z_]`
- `\D` is a negated `\d`; it represents any character but a digit `^[^0-9]`
- `\S` is a negated `\s`; it represents any non-whitespace character `^[^\s]`
- `\W` is a negated `\w`; it represents any non-word character `^[^\w]`
- The period `'.'` matches any character but `"\n"`

The `\d\s\w\D\S\W` abbreviations can be used both inside and outside of character classes. Here are some in use:

```

/\d\d:\d\d:\d\d/; # matches a hh:mm:ss time format
/[\d\s]/;         # matches any digit or whitespace character
/\w\W\w/;         # matches a word char, followed by a
                  # non-word char, followed by a word char
/..rt/;           # matches any two chars, followed by 'rt'
/end\./;          # matches 'end.'
/end[.]/;         # same thing, matches 'end.'
```

Because a period is a metacharacter, it needs to be escaped to match as an ordinary period. Because, for example, `\d` and `\w` are sets of characters, it is incorrect to think of `^[^d\w]` as `[\D\W]`; in fact `^[^d\w]` is the same as `^[^\w]`, which is the same as `[\W]`. Think DeMorgan's laws.

An anchor useful in basic regexps is the **word anchor** `\b`. This matches a boundary between a word character and a non-word character `\w\W` or `\W\w`:

```

$x = "Housecat catenates house and cat";
$x =~ /cat/;      # matches cat in 'housecat'
$x =~ /\bcat/;    # matches cat in 'catenates'
$x =~ /cat\b/;    # matches cat in 'housecat'
$x =~ /\bcat\b/;  # matches 'cat' at end of string
```

Note in the last example, the end of the string is considered a word boundary.

You might wonder why `'.'` matches everything but `"\n"` – why not every character? The reason is that often one is matching against lines and would like to ignore the newline characters. For instance, while the string `"\n"` represents one line, we would like to think of as empty. Then

```

""    =~ /^$/;      # matches
"\n"  =~ /^$/;      # matches, "\n" is ignored

""    =~ /. /;      # doesn't match; it needs a char
""    =~ /^. $/;     # doesn't match; it needs a char
"\n"  =~ /^. $/;     # doesn't match; it needs a char other than "\n"
"a"    =~ /^. $/;    # matches
"a\n"  =~ /^. $/;    # matches, ignores the "\n"
```

This behavior is convenient, because we usually want to ignore newlines when we count and match characters in a line. Sometimes, however, we want to keep track of newlines. We might even want `^` and `$` to anchor at the beginning and end of lines within the string, rather than just the beginning and end of the string. Perl allows us to choose between ignoring and paying attention to newlines by using the `//s` and `//m` modifiers. `//s` and `//m` stand for single line and multi-line and they determine whether a string is to be treated as one continuous string, or as a set of lines. The two modifiers affect two aspects of how the regexp is interpreted: 1) how the `'.'` character class is defined, and 2) where the anchors `^` and `$` are able to match. Here are the four possible combinations:

- no modifiers (/): Default behavior. `'.'` matches any character except `"\n"`. `^` matches only at the beginning of the string and `$` matches only at the end or before a newline at the end.
- `s` modifier (/s): Treat string as a single long line. `'.'` matches any character, even `"\n"`. `^` matches only at the beginning of the string and `$` matches only at the end or before a newline at the end.
- `m` modifier (/m): Treat string as a set of multiple lines. `'.'` matches any character except `"\n"`. `^` and `$` are able to match at the start or end of *any* line within the string.
- both `s` and `m` modifiers (/sm): Treat string as a single long line, but detect multiple lines. `'.'` matches any character, even `"\n"`. `^` and `$`, however, are able to match at the start or end of *any* line within the string.

Here are examples of `//s` and `//m` in action:

```
$x = "There once was a girl\nWho programmed in Perl\n";

$x =~ /^Who/;    # doesn't match, "Who" not at start of string
$x =~ /^Who/s;   # doesn't match, "Who" not at start of string
$x =~ /^Who/m;   # matches, "Who" at start of second line
$x =~ /^Who/sm;  # matches, "Who" at start of second line

$x =~ /girl.Who/; # doesn't match, "." doesn't match "\n"
$x =~ /girl.Who/s; # matches, "." matches "\n"
$x =~ /girl.Who/m; # doesn't match, "." doesn't match "\n"
$x =~ /girl.Who/sm; # matches, "." matches "\n"
```

Most of the time, the default behavior is what is want, but `//s` and `//m` are occasionally very useful. If `//m` is being used, the start of the string can still be matched with `\A` and the end of string can still be matched with the anchors `\Z` (matches both the end and the newline before, like `$`) , and `\z` (matches only the end):

```
$x =~ /^Who/m;    # matches, "Who" at start of second line
$x =~ /\AWho/m;   # doesn't match, "Who" is not at start of string

$x =~ /girl$/m;   # matches, "girl" at end of first line
$x =~ /girl\Z/m;  # doesn't match, "girl" is not at end of string

$x =~ /Perl\Z/m;  # matches, "Perl" is at newline before end
$x =~ /Perl\z/m;  # doesn't match, "Perl" is not at end of string
```

We now know how to create choices among classes of characters in a regexp. What about choices among words or character strings? Such choices are described in the next section.

Matching this or that

Sometimes we would like to our regexp to be able to match different possible words or character strings. This is accomplished by using the **alternation** metacharacter `|`. To match `dog` or `cat`, we form the regexp `dog|cat`. As before, perl will try to match the regexp at the earliest possible point in the string. At each character position, perl will first try to match the first alternative, `dog`. If `dog` doesn't match, perl will then try the next alternative, `cat`. If `cat` doesn't match either, then the match fails and perl moves to the next position in the string. Some examples:

```
"cats and dogs" =~ /cat|dog|bird/; # matches "cat"
"cats and dogs" =~ /dog|cat|bird/; # matches "cat"
```

Even though `dog` is the first alternative in the second regexp, `cat` is able to match earlier in the string.

```
"cats"          =~ /c|ca|cat|cats/; # matches "c"
"cats"          =~ /cats|cat|ca|c/;  # matches "cats"
```

Here, all the alternatives match at the first string position, so the first alternative is the one that matches. If some of the alternatives are truncations of the others, put the longest ones first to give them a chance to

match.

```
"cab" =~ /a|b|c/ # matches "c"
                # /a|b|c/ == /[abc]/
```

The last example points out that character classes are like alternations of characters. At a given character position, the first alternative that allows the regexp match to succeed will be the one that matches.

Grouping things and hierarchical matching

Alternation allows a regexp to choose among alternatives, but by itself it is unsatisfying. The reason is that each alternative is a whole regexp, but sometime we want alternatives for just part of a regexp. For instance, suppose we want to search for housecats or housekeepers. The regexp `housecat|housekeeper` fits the bill, but is inefficient because we had to type `house` twice. It would be nice to have parts of the regexp be constant, like `house`, and some parts have alternatives, like `cat|keeper`.

The **grouping** metacharacters `()` solve this problem. Grouping allows parts of a regexp to be treated as a single unit. Parts of a regexp are grouped by enclosing them in parentheses. Thus we could solve the `housecat|housekeeper` by forming the regexp as `house(cat|keeper)`. The regexp `house(cat|keeper)` means match `house` followed by either `cat` or `keeper`. Some more examples are

```
/(a|b)b/;      # matches 'ab' or 'bb'
/(ac|b)b/;     # matches 'acb' or 'bb'
/^(^a|b)c/;    # matches 'ac' at start of string or 'bc' anywhere
/(a|[bc])d/;   # matches 'ad', 'bd', or 'cd'

/house(cat|)/;  # matches either 'housecat' or 'house'
/house(cat(s|)|)/; # matches either 'housecats' or 'housecat' or
                  # 'house'. Note groups can be nested.

/(19|20|)\d\d/; # match years 19xx, 20xx, or the Y2K problem, xx
"20" =~ /(19|20|)\d\d/; # matches the null alternative '()\d\d',
                        # because '20\d\d' can't match
```

Alternations behave the same way in groups as out of them: at a given string position, the leftmost alternative that allows the regexp to match is taken. So in the last example at the first string position, "20" matches the second alternative, but there is nothing left over to match the next two digits `\d\d`. So perl moves on to the next alternative, which is the null alternative and that works, since "20" is two digits.

The process of trying one alternative, seeing if it matches, and moving on to the next alternative if it doesn't, is called **backtracking**. The term 'backtracking' comes from the idea that matching a regexp is like a walk in the woods. Successfully matching a regexp is like arriving at a destination. There are many possible trailheads, one for each string position, and each one is tried in order, left to right. From each trailhead there may be many paths, some of which get you there, and some which are dead ends. When you walk along a trail and hit a dead end, you have to backtrack along the trail to an earlier point to try another trail. If you hit your destination, you stop immediately and forget about trying all the other trails. You are persistent, and only if you have tried all the trails from all the trailheads and not arrived at your destination, do you declare failure. To be concrete, here is a step-by-step analysis of what perl does when it tries to match the regexp

```
"abcde" =~ /(abd|abc)(df|d|de)/;
```

- 0 Start with the first letter in the string 'a'.
- 1 Try the first alternative in the first group 'abd'.
- 2 Match 'a' followed by 'b'. So far so good.
- 3 'd' in the regexp doesn't match 'c' in the string – a dead end. So backtrack two characters and pick the second alternative in the first group 'abc'.

- 4 Match 'a' followed by 'b' followed by 'c'. We are on a roll and have satisfied the first group. Set \$1 to 'abc'.
- 5 Move on to the second group and pick the first alternative 'df'.
- 6 Match the 'd'.
- 7 'f' in the regexp doesn't match 'e' in the string, so a dead end. Backtrack one character and pick the second alternative in the second group 'd'.
- 8 'd' matches. The second grouping is satisfied, so set \$2 to 'd'.
- 9 We are at the end of the regexp, so we are done! We have matched 'abcd' out of the string "abcde".

There are a couple of things to note about this analysis. First, the third alternative in the second group 'de' also allows a match, but we stopped before we got to it – at a given character position, leftmost wins. Second, we were able to get a match at the first character position of the string 'a'. If there were no matches at the first position, perl would move to the second character position 'b' and attempt the match all over again. Only when all possible paths at all possible character positions have been exhausted does perl give up and declare `$string =~ /(abd|abc)(df|d|de)/;` to be false.

Even with all this work, regexp matching happens remarkably fast. To speed things up, during compilation stage, perl compiles the regexp into a compact sequence of opcodes that can often fit inside a processor cache. When the code is executed, these opcodes can then run at full throttle and search very quickly.

Extracting matches

The grouping metacharacters `()` also serve another completely different function: they allow the extraction of the parts of a string that matched. This is very useful to find out what matched and for text processing in general. For each grouping, the part that matched inside goes into the special variables `$1`, `$2`, etc. They can be used just as ordinary variables:

```
# extract hours, minutes, seconds
$time =~ /(\d\d):(\d\d):(\d\d)/; # match hh:mm:ss format
$hours = $1;
$minutes = $2;
$seconds = $3;
```

Now, we know that in scalar context, `$time =~ /(\d\d):(\d\d):(\d\d)/` returns a true or false value. In list context, however, it returns the list of matched values `($1, $2, $3)`. So we could write the code more compactly as

```
# extract hours, minutes, seconds
($hours, $minutes, $seconds) = ($time =~ /(\d\d):(\d\d):(\d\d)/);
```

If the groupings in a regexp are nested, `$1` gets the group with the leftmost opening parenthesis, `$2` the next opening parenthesis, etc. For example, here is a complex regexp and the matching variables indicated below it:

```
/(ab(cd|ef)(gi)|j)/;
 1  2      34
```

so that if the regexp matched, e.g., `$2` would contain 'cd' or 'ef'. For convenience, perl sets `$+` to the string held by the highest numbered `$1`, `$2`, ... that got assigned (and, somewhat related, `$_N` to the value of the `$1`, `$2`, ... most-recently assigned; i.e. the `$1`, `$2`, ... associated with the rightmost closing parenthesis used in the match).

Closely associated with the matching variables `$1`, `$2`, ... are the **backreferences** `\1`, `\2`, Backreferences are simply matching variables that can be used *inside* a regexp. This is a really nice feature – what matches later in a regexp can depend on what matched earlier in the regexp. Suppose we wanted to look for doubled words in text, like 'the the'. The following regexp finds all 3-letter doubles with a space in between:

```
/(\\w\\w\\w)\\s\\1/;
```

The grouping assigns a value to `\\1`, so that the same 3 letter sequence is used for both parts. Here are some words with repeated parts:

```
% simple_grep '^((\\w\\w\\w\\w|\\w\\w\\w|\\w\\w|\\w)\\1$' /usr/dict/words
beriberi
booboo
coco
mama
murmur
papa
```

The regexp has a single grouping which considers 4-letter combinations, then 3-letter combinations, etc. and uses `\\1` to look for a repeat. Although `$1` and `\\1` represent the same thing, care should be taken to use matched variables `$1`, `$2`, ... only outside a regexp and backreferences `\\1`, `\\2`, ... only inside a regexp; not doing so may lead to surprising and/or undefined results.

In addition to what was matched, Perl 5.6.0 also provides the positions of what was matched with the `@-` and `@+` arrays. `$-[0]` is the position of the start of the entire match and `$+[0]` is the position of the end. Similarly, `$-[n]` is the position of the start of the `$n` match and `$+[n]` is the position of the end. If `$n` is undefined, so are `$-[n]` and `$+[n]`. Then this code

```
$x = "Mmm...donut, thought Homer";
$x =~ /^(Mmm|Yech)\\.\\.\\. (donut|peas)/; # matches
foreach $expr (1..$#-) {
    print "Match $expr: '${$expr}' at position ($-[ $expr ], $+[ $expr ])\n";
}
```

prints

```
Match 1: 'Mmm' at position (0,3)
Match 2: 'donut' at position (6,11)
```

Even if there are no groupings in a regexp, it is still possible to find out what exactly matched in a string. If you use them, perl will set `$'` to the part of the string before the match, will set `$&` to the part of the string that matched, and will set `$'` to the part of the string after the match. An example:

```
$x = "the cat caught the mouse";
$x =~ /cat/; # $' = 'the ', $& = 'cat', $' = ' caught the mouse'
$x =~ /the/; # $' = '', $& = 'the', $' = ' cat caught the mouse'
```

In the second match, `$' = ''` because the regexp matched at the first character position in the string and stopped, it never saw the second 'the'. It is important to note that using `$'` and `$'` slows down regexp matching quite a bit, and `$&` slows it down to a lesser extent, because if they are used in one regexp in a program, they are generated for <all> regexps in the program. So if raw performance is a goal of your application, they should be avoided. If you need them, use `@-` and `@+` instead:

```
$' is the same as substr( $x, 0, $-[0] )
$& is the same as substr( $x, $-[0], $+[0]-$-[0] )
$' is the same as substr( $x, $+[0] )
```

Matching repetitions

The examples in the previous section display an annoying weakness. We were only matching 3-letter words, or syllables of 4 letters or less. We'd like to be able to match words or syllables of any length, without writing out tedious alternatives like `\\w\\w\\w\\w|\\w\\w\\w|\\w\\w|\\w`.

This is exactly the problem the **quantifier** metacharacters `?`, `*`, `+`, and `{ }` were created for. They allow us to determine the number of repeats of a portion of a regexp we consider to be a match. Quantifiers are put immediately after the character, character class, or grouping that we want to specify. They have the following meanings:

- `a?` = match 'a' 1 or 0 times
- `a*` = match 'a' 0 or more times, i.e., any number of times
- `a+` = match 'a' 1 or more times, i.e., at least once
- `a{n,m}` = match at least `n` times, but not more than `m` times.
- `a{n,}` = match at least `n` or more times
- `a{n}` = match exactly `n` times

Here are some examples:

```
[a-z]+\s+\d*/; # match a lowercase word, at least some space, and
                # any number of digits
/(\w+)\s+\1/;   # match doubled words of arbitrary length
/Y(es)?/i;      # matches 'y', 'Y', or a case-insensitive 'yes'
$year =~ /\d{2,4}/; # make sure year is at least 2 but not more
                  # than 4 digits
$year =~ /\d{4}|\d{2}/; # better match; throw out 3 digit dates
$year =~ /\d{2}(\d{2})?/; # same thing written differently. However,
                          # this produces $1 and the other does not.

% simple_grep '^(\w+)\l$' /usr/dict/words # isn't this easier?
beriberi
booboo
coco
mama
murmur
papa
```

For all of these quantifiers, perl will try to match as much of the string as possible, while still allowing the regexp to succeed. Thus with `/a?...`, perl will first try to match the regexp with the `a` present; if that fails, perl will try to match the regexp without the `a` present. For the quantifier `*`, we get the following:

```
$x = "the cat in the hat";
$x =~ /^(.*) (cat) (.*)$/; # matches,
                          # $1 = 'the '
                          # $2 = 'cat'
                          # $3 = ' in the hat'
```

Which is what we might expect, the match finds the only cat in the string and locks onto it. Consider, however, this regexp:

```
$x =~ /^(.*) (at) (.*)$/; # matches,
                          # $1 = 'the cat in the h'
                          # $2 = 'at'
                          # $3 = '' (0 matches)
```

One might initially guess that perl would find the `at` in `cat` and stop there, but that wouldn't give the longest possible string to the first quantifier `.*`. Instead, the first quantifier `.*` grabs as much of the string as possible while still having the regexp match. In this example, that means having the `at` sequence with the final `at` in the string. The other important principle illustrated here is that when there are two or more elements in a regexp, the *leftmost* quantifier, if there is one, gets to grab as much the string as possible, leaving the rest of the regexp to fight over scraps. Thus in our example, the first quantifier `.*` grabs most of the string, while the second quantifier `.*` gets the empty string. Quantifiers that grab as much of the string as possible are called **maximal match** or **greedy** quantifiers.

When a regexp can match a string in several different ways, we can use the principles above to predict which way the regexp will match:

- Principle 0: Taken as a whole, any regexp will be matched at the earliest possible position in the string.
- Principle 1: In an alternation `a|b|c...`, the leftmost alternative that allows a match for the whole regexp will be the one used.
- Principle 2: The maximal matching quantifiers `?`, `*`, `+` and `{n,m}` will in general match as much of the string as possible while still allowing the whole regexp to match.
- Principle 3: If there are two or more elements in a regexp, the leftmost greedy quantifier, if any, will match as much of the string as possible while still allowing the whole regexp to match. The next leftmost greedy quantifier, if any, will try to match as much of the string remaining available to it as possible, while still allowing the whole regexp to match. And so on, until all the regexp elements are satisfied.

As we have seen above, Principle 0 overrides the others – the regexp will be matched as early as possible, with the other principles determining how the regexp matches at that earliest character position.

Here is an example of these principles in action:

```
$x = "The programming republic of Perl";
$x =~ /^(.+)(e|r)(.*)$/; # matches,
                        # $1 = 'The programming republic of Pe'
                        # $2 = 'r'
                        # $3 = 'l'
```

This regexp matches at the earliest string position, `'T'`. One might think that `e`, being leftmost in the alternation, would be matched, but `r` produces the longest string in the first quantifier.

```
$x =~ /(m{1,2})(.*)$/; # matches,
                        # $1 = 'mm'
                        # $2 = 'ing republic of Perl'
```

Here, The earliest possible match is at the first `'m'` in `programming`. `m{1,2}` is the first quantifier, so it gets to match a maximal `mm`.

```
$x =~ /.*(m{1,2})(.*)$/; # matches,
                        # $1 = 'm'
                        # $2 = 'ing republic of Perl'
```

Here, the regexp matches at the start of the string. The first quantifier `.*` grabs as much as possible, leaving just a single `'m'` for the second quantifier `m{1,2}`.

```
$x =~ /(.*?) (m{1,2})(.*)$/; # matches,
                        # $1 = 'a'
                        # $2 = 'mm'
                        # $3 = 'ing republic of Perl'
```

Here, `.*?` eats its maximal one character at the earliest possible position in the string, `'a'` in `programming`, leaving `m{1,2}` the opportunity to match both `m`'s. Finally,

```
"aXXXb" =~ /(X*)/; # matches with $1 = ''
```

because it can match zero copies of `'X'` at the beginning of the string. If you definitely want to match at least one `'X'`, use `X+`, not `X*`.

Sometimes greed is not good. At times, we would like quantifiers to match a *minimal* piece of string, rather than a maximal piece. For this purpose, Larry Wall created the **minimal match** or **non-greedy** quantifiers `??`, `*?`, `++`, and `{ }?`. These are the usual quantifiers with a `?` appended to them. They have the following meanings:

- `a??` = match 'a' 0 or 1 times. Try 0 first, then 1.
- `a*` = match 'a' 0 or more times, i.e., any number of times, but as few times as possible
- `a+?` = match 'a' 1 or more times, i.e., at least once, but as few times as possible
- `a{n,m}?` = match at least `n` times, not more than `m` times, as few times as possible
- `a{n,}?` = match at least `n` times, but as few times as possible
- `a{n}?` = match exactly `n` times. Because we match exactly `n` times, `a{n}?` is equivalent to `a{n}` and is just there for notational consistency.

Let's look at the example above, but with minimal quantifiers:

```
$x = "The programming republic of Perl";
$x =~ /^(.+?) (e|r) (.*)$/; # matches,
                             # $1 = 'Th'
                             # $2 = 'e'
                             # $3 = ' programming republic of Perl'
```

The minimal string that will allow both the start of the string `^` and the alternation to match is `Th`, with the alternation `e|r` matching `e`. The second quantifier `.*` is free to gobble up the rest of the string.

```
$x =~ /(m{1,2}?) (.*)$/; # matches,
                          # $1 = 'm'
                          # $2 = 'ming republic of Perl'
```

The first string position that this regexp can match is at the first 'm' in `programming`. At this position, the minimal `m{1,2}?` matches just one 'm'. Although the second quantifier `.*?` would prefer to match no characters, it is constrained by the end-of-string anchor `$` to match the rest of the string.

```
$x =~ /(.*?) (m{1,2}?) (.*)$/; # matches,
                              # $1 = 'The progra'
                              # $2 = 'm'
                              # $3 = 'ming republic of Perl'
```

In this regexp, you might expect the first minimal quantifier `.*?` to match the empty string, because it is not constrained by a `^` anchor to match the beginning of the word. Principle 0 applies here, however. Because it is possible for the whole regexp to match at the start of the string, it *will* match at the start of the string. Thus the first quantifier has to match everything up to the first `m`. The second minimal quantifier matches just one `m` and the third quantifier matches the rest of the string.

```
$x =~ /(.*?) (m{1,2}) (.*)$/; # matches,
                              # $1 = 'a'
                              # $2 = 'mm'
                              # $3 = 'ing republic of Perl'
```

Just as in the previous regexp, the first quantifier `.*?` can match earliest at position 'a', so it does. The second quantifier is greedy, so it matches `mm`, and the third matches the rest of the string.

We can modify principle 3 above to take into account non-greedy quantifiers:

- Principle 3: If there are two or more elements in a regexp, the leftmost greedy (non-greedy) quantifier, if any, will match as much (little) of the string as possible while still allowing the whole regexp to match. The next leftmost greedy (non-greedy) quantifier, if any, will try to match as much (little) of the string remaining available to it as possible, while still allowing the whole regexp to match. And so on, until all the regexp elements are satisfied.

Just like alternation, quantifiers are also susceptible to backtracking. Here is a step-by-step analysis of the example

```
$x = "the cat in the hat";
$x =~ /^(.*) (at) (.*)$/; # matches,
                        # $1 = 'the cat in the h'
                        # $2 = 'at'
                        # $3 = ''      (0 matches)
```

- 0 Start with the first letter in the string 't'.
- 1 The first quantifier '.*' starts out by matching the whole string 'the cat in the hat'.
- 2 'a' in the regexp element 'at' doesn't match the end of the string. Backtrack one character.
- 3 'a' in the regexp element 'at' still doesn't match the last letter of the string 't', so backtrack one more character.
- 4 Now we can match the 'a' and the 't'.
- 5 Move on to the third element '.*'. Since we are at the end of the string and '.*' can match 0 times, assign it the empty string.
- 6 We are done!

Most of the time, all this moving forward and backtracking happens quickly and searching is fast. There are some pathological regexps, however, whose execution time exponentially grows with the size of the string. A typical structure that blows up in your face is of the form

```
/(a|b+)*;/
```

The problem is the nested indeterminate quantifiers. There are many different ways of partitioning a string of length n between the $+$ and $*$: one repetition with $b+$ of length n , two repetitions with the first $b+$ length k and the second with length $n-k$, m repetitions whose bits add up to length n , etc. In fact there are an exponential number of ways to partition a string as a function of length. A regexp may get lucky and match early in the process, but if there is no match, perl will try *every* possibility before giving up. So be careful with nested $*$'s, $\{n,m\}$'s, and $+$'s. The book *Mastering regular expressions* by Jeffrey Friedl gives a wonderful discussion of this and other efficiency issues.

Building a regexp

At this point, we have all the basic regexp concepts covered, so let's give a more involved example of a regular expression. We will build a regexp that matches numbers.

The first task in building a regexp is to decide what we want to match and what we want to exclude. In our case, we want to match both integers and floating point numbers and we want to reject any string that isn't a number.

The next task is to break the problem down into smaller problems that are easily converted into a regexp.

The simplest case is integers. These consist of a sequence of digits, with an optional sign in front. The digits we can represent with $\backslash d+$ and the sign can be matched with $[+-]$. Thus the integer regexp is

```
/[+-]?\d+/; # matches integers
```

A floating point number potentially has a sign, an integral part, a decimal point, a fractional part, and an exponent. One or more of these parts is optional, so we need to check out the different possibilities. Floating point numbers which are in proper form include 123., 0.345, .34, -1e6, and 25.4E-72. As with integers, the sign out front is completely optional and can be matched by $[+-]?$. We can see that if there is no exponent, floating point numbers must have a decimal point, otherwise they are integers. We might be tempted to model these with $\backslash d*\.\backslash d*$, but this would also match just a single decimal point, which is not a number. So the three cases of floating point number sans exponent are

```
/[+-]?\d+\./; # 1., 321., etc.
/[+-]?\.\d+/; # .1, .234, etc.
/[+-]?\d+\.\d+/; # 1.0, 30.56, etc.
```

These can be combined into a single regexp with a three-way alternation:

```
/[+-]?(\d+\.\d+|\d+\.|\.\d+)/; # floating point, no exponent
```

In this alternation, it is important to put `\d+\.\d+` before `\d+\.`. If `\d+\.` were first, the regexp would happily match that and ignore the fractional part of the number.

Now consider floating point numbers with exponents. The key observation here is that *both* integers and numbers with decimal points are allowed in front of an exponent. Then exponents, like the overall sign, are independent of whether we are matching numbers with or without decimal points, and can be ‘decoupled’ from the mantissa. The overall form of the regexp now becomes clear:

```
/^(optional sign)(integer | f.p. mantissa)(optional exponent)$/;
```

The exponent is an `e` or `E`, followed by an integer. So the exponent regexp is

```
/[eE][+-]?\d+/; # exponent
```

Putting all the parts together, we get a regexp that matches numbers:

```
/^[+-]?(\d+\.\d+|\d+\.|\.\d+|\d+)([eE][+-]?\d+)?$/; # Ta da!
```

Long regexps like this may impress your friends, but can be hard to decipher. In complex situations like this, the `/x` modifier for a match is invaluable. It allows one to put nearly arbitrary whitespace and comments into a regexp without affecting their meaning. Using it, we can rewrite our ‘extended’ regexp in the more pleasing form

```
/^
  [+-]?          # first, match an optional sign
  (
    \d+\.\d+     # then match integers or f.p. mantissas:
    | \d+\.      # mantissa of the form a.b
    | \.\d+      # mantissa of the form a.
    | \d+        # mantissa of the form .b
    | \d+        # integer of the form a
  )
  ([eE][+-]?\d+)? # finally, optionally match an exponent
$/x;
```

If whitespace is mostly irrelevant, how does one include space characters in an extended regexp? The answer is to backslash it `\\` or put it in a character class `[]`. The same thing goes for pound signs, use `\\#` or `[#]`. For instance, Perl allows a space between the sign and the mantissa/integer, and we could add this to our regexp as follows:

```
/^
  [+-]?\\ *      # first, match an optional sign *and space*
  (
    \d+\.\d+     # then match integers or f.p. mantissas:
    | \d+\.      # mantissa of the form a.b
    | \.\d+      # mantissa of the form a.
    | \d+        # mantissa of the form .b
    | \d+        # integer of the form a
  )
  ([eE][+-]?\\d+)? # finally, optionally match an exponent
$/x;
```

In this form, it is easier to see a way to simplify the alternation. Alternatives 1, 2, and 4 all start with `\d+`, so it could be factored out:

```
/^
  [+-]?\\ *      # first, match an optional sign
  (
    \d+          # then match integers or f.p. mantissas:
    | \d+        # start out with a ...
  )
```

```

        (
            \.d* # mantissa of the form a.b or a.
        )?      # ? takes care of integers of the form a
        | \.d+  # mantissa of the form .b
    )
    ([eE] [+−]?d+)? # finally, optionally match an exponent
$/x;

```

or written in the compact form,

```
/^[+−]? \* (\d+(\.d*)?|\.d+) ([eE] [+−]?d+)?$/;
```

This is our final regexp. To recap, we built a regexp by

- specifying the task in detail,
- breaking down the problem into smaller parts,
- translating the small parts into regexps,
- combining the regexps,
- and optimizing the final combined regexp.

These are also the typical steps involved in writing a computer program. This makes perfect sense, because regular expressions are essentially programs written a little computer language that specifies patterns.

Using regular expressions in Perl

The last topic of Part 1 briefly covers how regexps are used in Perl programs. Where do they fit into Perl syntax?

We have already introduced the matching operator in its default `/regexp/` and arbitrary delimiter `m!regexp!` forms. We have used the binding operator `=~` and its negation `!~` to test for string matches. Associated with the matching operator, we have discussed the single line `/s`, multi-line `/m`, case-insensitive `/i` and extended `/x` modifiers.

There are a few more things you might want to know about matching operators. First, we pointed out earlier that variables in regexps are substituted before the regexp is evaluated:

```

$pattern = 'Seuss';
while (<>) {
    print if /$pattern/;
}

```

This will print any lines containing the word `Seuss`. It is not as efficient as it could be, however, because perl has to re-evaluate `$pattern` each time through the loop. If `$pattern` won't be changing over the lifetime of the script, we can add the `/o` modifier, which directs perl to only perform variable substitutions once:

```

#!/usr/bin/perl
# Improved simple_grep
$regexp = shift;
while (<>) {
    print if /$regexp/o; # a good deal faster
}

```

If you change `$pattern` after the first substitution happens, perl will ignore it. If you don't want any substitutions at all, use the special delimiter `m''`:

```

$pattern = 'Seuss';
while (<>) {
    print if m'$pattern'; # matches '$pattern', not 'Seuss'
}

```


`m''` acts like single quotes on a regexp; all other `m` delimiters act like double quotes. If the regexp evaluates to the empty string, the regexp in the *last successful match* is used instead. So we have

```
"dog" =~ /d/; # 'd' matches
"dogbert" =~ //; # this matches the 'd' regexp used before
```

The final two modifiers `//g` and `//c` concern multiple matches. The modifier `//g` stands for global matching and allows the matching operator to match within a string as many times as possible. In scalar context, successive invocations against a string will have `//g` jump from match to match, keeping track of position in the string as it goes along. You can get or set the position with the `pos()` function.

The use of `//g` is shown in the following example. Suppose we have a string that consists of words separated by spaces. If we know how many words there are in advance, we could extract the words using groupings:

```
$x = "cat dog house"; # 3 words
$x =~ /^s*(\w+)\s+(\w+)\s+(\w+)\s*$/; # matches,
                                     # $1 = 'cat'
                                     # $2 = 'dog'
                                     # $3 = 'house'
```

But what if we had an indeterminate number of words? This is the sort of task `//g` was made for. To extract all words, form the simple regexp `(\w+)` and loop over all matches with `/(\w+)/g`:

```
while ($x =~ /(\w+)/g) {
    print "Word is $1, ends at position ", pos $x, "\n";
}
```

prints

```
Word is cat, ends at position 3
Word is dog, ends at position 7
Word is house, ends at position 13
```

A failed match or changing the target string resets the position. If you don't want the position reset after failure to match, add the `//c`, as in `/regexp/gc`. The current position in the string is associated with the string, not the regexp. This means that different strings have different positions and their respective positions can be set or read independently.

In list context, `//g` returns a list of matched groupings, or if there are no groupings, a list of matches to the whole regexp. So if we wanted just the words, we could use

```
@words = ($x =~ /(\w+)/g); # matches,
                          # $word[0] = 'cat'
                          # $word[1] = 'dog'
                          # $word[2] = 'house'
```

Closely associated with the `//g` modifier is the `\G` anchor. The `\G` anchor matches at the point where the previous `//g` match left off. `\G` allows us to easily do context-sensitive matching:

```
$metric = 1; # use metric units
...
$x = <FILE>; # read in measurement
$x =~ /^([+-]?\d+)\s*/g; # get magnitude
$weight = $1;
if ($metric) { # error checking
    print "Units error!" unless $x =~ /\Gkg\./g;
}
else {
    print "Units error!" unless $x =~ /\Glbs\./g;
}
```

```
$x =~ /\G\s+(widget|sprocket)/g; # continue processing
```

The combination of `//g` and `\G` allows us to process the string a bit at a time and use arbitrary Perl logic to decide what to do next. Currently, the `\G` anchor is only fully supported when used to anchor to the start of the pattern.

`\G` is also invaluable in processing fixed length records with regexps. Suppose we have a snippet of coding region DNA, encoded as base pair letters ATCGTTGAAT... and we want to find all the stop codons TGA. In a coding region, codons are 3-letter sequences, so we can think of the DNA snippet as a sequence of 3-letter records. The naive regexp

```
# expanded, this is "ATC GTT GAA TGC AAA TGA CAT GAC"
$dna = "ATCGTTGAATGCAAATGACATGAC";
$dna =~ /TGA/;
```

doesn't work; it may match a TGA, but there is no guarantee that the match is aligned with codon boundaries, e.g., the substring GTT GAA gives a match. A better solution is

```
while ($dna =~ /(\w\w\w)*?TGA/g) { # note the minimal *?
    print "Got a TGA stop codon at position ", pos $dna, "\n";
}
```

which prints

```
Got a TGA stop codon at position 18
Got a TGA stop codon at position 23
```

Position 18 is good, but position 23 is bogus. What happened?

The answer is that our regexp works well until we get past the last real match. Then the regexp will fail to match a synchronized TGA and start stepping ahead one character position at a time, not what we want. The solution is to use `\G` to anchor the match to the codon alignment:

```
while ($dna =~ /\G(\w\w\w)*?TGA/g) {
    print "Got a TGA stop codon at position ", pos $dna, "\n";
}
```

This prints

```
Got a TGA stop codon at position 18
```

which is the correct answer. This example illustrates that it is important not only to match what is desired, but to reject what is not desired.

search and replace

Regular expressions also play a big role in **search and replace** operations in Perl. Search and replace is accomplished with the `s///` operator. The general form is `s/regexp/replacement/modifiers`, with everything we know about regexps and modifiers applying in this case as well. The replacement is a Perl double quoted string that replaces in the string whatever is matched with the regexp. The operator `=~` is also used here to associate a string with `s///`. If matching against `$_`, the `$_ =~` can be dropped. If there is a match, `s///` returns the number of substitutions made, otherwise it returns false. Here are a few examples:

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/; # $x contains "Time to feed the hacker!"
if ($x =~ s/^(Time.*hacker)!$/! now!/) {
    $more_insistent = 1;
}
$y = "'quoted words'";
$y =~ s/^(.*)'$/!$/; # strip single quotes,
# $y contains "quoted words"
```

In the last example, the whole string was matched, but only the part inside the single quotes was grouped. With the `s///` operator, the matched variables `$1`, `$2`, etc. are immediately available for use in the replacement expression, so we use `$1` to replace the quoted string with just what was quoted. With the global modifier, `s///g` will search and replace all occurrences of the regexp in the string:

```
$x = "I batted 4 for 4";
$x =~ s/4/four/;      # doesn't do it all:
                       # $x contains "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g;     # does it all:
                       # $x contains "I batted four for four"
```

If you prefer ‘`regex`’ over ‘`regexp`’ in this tutorial, you could use the following program to replace it:

```
% cat > simple_replace
#!/usr/bin/perl
$regexp = shift;
$replacement = shift;
while (<>) {
    s/$regexp/$replacement/go;
    print;
}
^D

% simple_replace regexp regex perlretut.pod
```

In `simple_replace` we used the `s///g` modifier to replace all occurrences of the regexp on each line and the `s///o` modifier to compile the regexp only once. As with `simple_grep`, both the `print` and the `s/$regexp/$replacement/go` use `$_` implicitly.

A modifier available specifically to search and replace is the `s///e` evaluation modifier. `s///e` wraps an `eval{...}` around the replacement string and the evaluated result is substituted for the matched substring. `s///e` is useful if you need to do a bit of computation in the process of replacing text. This example counts character frequencies in a line:

```
$x = "Bill the cat";
$x =~ s/(.)/$chars{$1}++;$1/eg; # final $1 replaces char with itself
print "frequency of '$_' is $chars{$_}\n"
      foreach (sort {$chars{$b} <=> $chars{$a}} keys %chars);
```

This prints

```
frequency of ' ' is 2
frequency of 't' is 2
frequency of 'l' is 2
frequency of 'B' is 1
frequency of 'c' is 1
frequency of 'e' is 1
frequency of 'h' is 1
frequency of 'i' is 1
frequency of 'a' is 1
```

As with the `match m//` operator, `s///` can use other delimiters, such as `s!!!` and `s{ }{ }`, and even `s{ }//`. If single quotes are used `s` ```, then the regexp and replacement are treated as single quoted strings and there are no substitutions. `s///` in list context returns the same thing as in scalar context, i.e., the number of matches.

The split operator

The `split` function can also optionally use a matching operator `m//` to split a string. `split`

`/regex/, string, limit` splits `string` into a list of substrings and returns that list. The `regex` is used to match the character sequence that the `string` is split with respect to. The `limit`, if present, constrains splitting into no more than `limit` number of strings. For example, to split a string into words, use

```
$x = "Calvin and Hobbes";
@words = split /\s+/, $x; # $word[0] = 'Calvin'
                          # $word[1] = 'and'
                          # $word[2] = 'Hobbes'
```

If the empty `regex` `//` is used, the `regex` always matches and the string is split into individual characters. If the `regex` has groupings, then list produced contains the matched substrings from the groupings as well. For instance,

```
$x = "/usr/bin/perl";
@dirs = split m!/!, $x; # $dirs[0] = ''
                       # $dirs[1] = 'usr'
                       # $dirs[2] = 'bin'
                       # $dirs[3] = 'perl'
@parts = split m!(/)! , $x; # $parts[0] = ''
                           # $parts[1] = '/'
                           # $parts[2] = 'usr'
                           # $parts[3] = '/'
                           # $parts[4] = 'bin'
                           # $parts[5] = '/'
                           # $parts[6] = 'perl'
```

Since the first character of `$x` matched the `regex`, `split` prepended an empty initial element to the list.

If you have read this far, congratulations! You now have all the basic tools needed to use regular expressions to solve a wide range of text processing problems. If this is your first time through the tutorial, why not stop here and play around with `regexps` a while... Part 2 concerns the more esoteric aspects of regular expressions and those concepts certainly aren't needed right at the start.

Part 2: Power tools

OK, you know the basics of `regexps` and you want to know more. If matching regular expressions is analogous to a walk in the woods, then the tools discussed in Part 1 are analogous to topo maps and a compass, basic tools we use all the time. Most of the tools in part 2 are analogous to flare guns and satellite phones. They aren't used too often on a hike, but when we are stuck, they can be invaluable.

What follows are the more advanced, less used, or sometimes esoteric capabilities of `perl` `regexps`. In Part 2, we will assume you are comfortable with the basics and concentrate on the new features.

More on characters, strings, and character classes

There are a number of escape sequences and character classes that we haven't covered yet.

There are several escape sequences that convert characters or strings between upper and lower case. `\l` and `\u` convert the next character to lower or upper case, respectively:

```
$x = "perl";
$string =~ /\u$x/; # matches 'Perl' in $string
$x = "M(rs?|s)\."; # note the double backslash
$string =~ /\l$x/; # matches 'mr.', 'mrs.', and 'ms.',
```

`\L` and `\U` converts a whole substring, delimited by `\L` or `\U` and `\E`, to lower or upper case:

```
$x = "This word is in lower case:\L SHOUT\E";
$x =~ /shout/; # matches
$x = "I STILL KEYPUNCH CARDS FOR MY 360"
$x =~ /\Ukeypunch/; # matches punch card string
```

If there is no `\E`, case is converted until the end of the string. The regexps `\L\u$word` or `\u\L$word` convert the first character of `$word` to uppercase and the rest of the characters to lowercase.

Control characters can be escaped with `\c`, so that a control-Z character would be matched with `\cZ`. The escape sequence `\Q...\E` quotes, or protects most non-alphabetic characters. For instance,

```
$x = "\QThat !^*&%~& cat!";
$x =~ /\Q!^*&%~&\E/; # check for rough language
```

It does not protect `$` or `@`, so that variables can still be substituted.

With the advent of 5.6.0, perl regexps can handle more than just the standard ASCII character set. Perl now supports **Unicode**, a standard for encoding the character sets from many of the world's written languages. Unicode does this by allowing characters to be more than one byte wide. Perl uses the UTF-8 encoding, in which ASCII characters are still encoded as one byte, but characters greater than `chr(127)` may be stored as two or more bytes.

What does this mean for regexps? Well, regexp users don't need to know much about perl's internal representation of strings. But they do need to know 1) how to represent Unicode characters in a regexp and 2) when a matching operation will treat the string to be searched as a sequence of bytes (the old way) or as a sequence of Unicode characters (the new way). The answer to 1) is that Unicode characters greater than `chr(127)` may be represented using the `\x{hex}` notation, with `hex` a hexadecimal integer:

```
/\x{263a}/; # match a Unicode smiley face :)
```

Unicode characters in the range of 128–255 use two hexadecimal digits with braces: `\x{ab}`. Note that this is different than `\xab`, which is just a hexadecimal byte with no Unicode significance.

NOTE: in Perl 5.6.0 it used to be that one needed to say `use utf8` to use any Unicode features. This is no more the case: for almost all Unicode processing, the explicit `utf8` pragma is not needed. (The only case where it matters is if your Perl script is in Unicode and encoded in UTF-8, then an explicit `use utf8` is needed.)

Figuring out the hexadecimal sequence of a Unicode character you want or deciphering someone else's hexadecimal Unicode regexp is about as much fun as programming in machine code. So another way to specify Unicode characters is to use the **named character** escape sequence `\N{name}`. `name` is a name for the Unicode character, as specified in the Unicode standard. For instance, if we wanted to represent or match the astrological sign for the planet Mercury, we could use

```
use charnames ":full"; # use named chars with Unicode full names
$x = "abc\N{MERCURY}def";
$x =~ /\N{MERCURY}/; # matches
```

One can also use short names or restrict names to a certain alphabet:

```
use charnames ':full';
print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";

use charnames ":short";
print "\N{greek:Sigma} is an upper-case sigma.\n";

use charnames qw(greek);
print "\N{sigma} is Greek sigma\n";
```

A list of full names is found in the file `Names.txt` in the `lib/perl5/5.X.X/unicore` directory.

The answer to requirement 2), as of 5.6.0, is that if a regexp contains Unicode characters, the string is searched as a sequence of Unicode characters. Otherwise, the string is searched as a sequence of bytes. If the string is being searched as a sequence of Unicode characters, but matching a single byte is required, we can use the `\C` escape sequence. `\C` is a character class akin to `.` except that it matches *any* byte 0–255. So

```
use charnames ":full"; # use named chars with Unicode full names
$x = "a";
```

```

$x =~ /\C/; # matches 'a', eats one byte
$x = "";
$x =~ /\C/; # doesn't match, no bytes to match
$x = "\N{MERCURY}"; # two-byte Unicode character
$x =~ /\C/; # matches, but dangerous!

```

The last regexp matches, but is dangerous because the string *character* position is no longer synchronized to the string *byte* position. This generates the warning ‘Malformed UTF-8 character’. `\C` is best used for matching the binary data in strings with binary data intermixed with Unicode characters.

Let us now discuss the rest of the character classes. Just as with Unicode characters, there are named Unicode character classes represented by the `\p{name}` escape sequence. Closely associated is the `\P{name}` character class, which is the negation of the `\p{name}` class. For example, to match lower and uppercase characters,

```

use charnames ":full"; # use named chars with Unicode full names
$x = "BOB";
$x =~ /\^\p{IsUpper}/; # matches, uppercase char class
$x =~ /\^\P{IsUpper}/; # doesn't match, char class sans uppercase
$x =~ /\^\p{IsLower}/; # doesn't match, lowercase char class
$x =~ /\^\P{IsLower}/; # matches, char class sans lowercase

```

Here is the association between some Perl named classes and the traditional Unicode classes:

Perl class name	Unicode class name or regular expression
IsAlpha	<code>/^[LM]/</code>
IsAlnum	<code>/^[LMN]/</code>
IsASCII	<code>\$code <= 127</code>
IsCntrl	<code>/^C/</code>
IsBlank	<code>\$code =~ /^(0020 0009)\$/ /^Z[^\p]/</code>
IsDigit	<code>Nd</code>
IsGraph	<code>/^([LMNPS] Co)/</code>
IsLower	<code>Ll</code>
IsPrint	<code>/^([LMNPS] Co Zs)/</code>
IsPunct	<code>/^P/</code>
IsSpace	<code>/^Z/ (\$code =~ /^(0009 000A 000B 000C 000D)\$/</code>
IsSpacePerl	<code>/^Z/ (\$code =~ /^(0009 000A 000C 000D 0085 2028 2029)\$/</code>
IsUpper	<code>/^L[ut]/</code>
IsWord	<code>/^[LMN]/ \$code eq "005F"</code>
IsXDigit	<code>\$code =~ /^00(3[0-9] [46][1-6])\$/</code>

You can also use the official Unicode class names with the `\p` and `\P`, like `\p{L}` for Unicode ‘letters’, or `\p{Lu}` for uppercase letters, or `\P{Nd}` for non-digits. If a name is just one letter, the braces can be dropped. For instance, `\pM` is the character class of Unicode ‘marks’, for example accent marks. For the full list see [perlunicode](#).

The Unicode has also been separated into various sets of characters which you can test with `\p{In...}` (in) and `\P{In...}` (not in), for example `\p{Latin}`, `\p{Greek}`, or `\P{Katakana}`. For the full list see [perlunicode](#).

`\X` is an abbreviation for a character class sequence that includes the Unicode ‘combining character sequences’. A ‘combining character sequence’ is a base character followed by any number of combining characters. An example of a combining character is an accent. Using the Unicode full names, e.g., `A + COMBINING RING` is a combining character sequence with base character `A` and combining character `COMBINING RING`, which translates in Danish to `A` with the circle atop it, as in the word `Angstrom`. `\X` is equivalent to `\PM\p*`, i.e., a non-mark followed by one or more marks.

For the full and latest information about Unicode see the latest Unicode standard, or the Unicode

Consortium's website <http://www.unicode.org/>

As if all those classes weren't enough, Perl also defines POSIX style character classes. These have the form `[:name:]`, with `name` the name of the POSIX class. The POSIX classes are `alpha`, `alnum`, `ascii`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`, and two extensions, `word` (a Perl extension to match `\w`), and `blank` (a GNU extension). If `utf8` is being used, then these classes are defined the same as their corresponding perl Unicode classes: `[:upper:]` is the same as `\p{IsUpper}`, etc. The POSIX character classes, however, don't require using `utf8`. The `[:digit:]`, `[:word:]`, and `[:space:]` correspond to the familiar `\d`, `\w`, and `\s` character classes. To negate a POSIX class, put a `^` in front of the name, so that, e.g., `[:^digit:]` corresponds to `\D` and under `utf8`, `\P{IsDigit}`. The Unicode and POSIX character classes can be used just like `\d`, with the exception that POSIX character classes can only be used inside of a character class:

```
/\s+[abc[:digit:]]xyz\s*/; # match a,b,c,x,y,z, or a digit
/^=item\s[[:digit:]]/;    # match '=item',
                           # followed by a space and a digit

use charnames ":full";
/\s+[abc\p{IsDigit}]xyz\s+/; # match a,b,c,x,y,z, or a digit
/^=item\s\p{IsDigit}/;      # match '=item',
                           # followed by a space and a digit
```

Whew! That is all the rest of the characters and character classes.

Compiling and saving regular expressions

In Part 1 we discussed the `//o` modifier, which compiles a regexp just once. This suggests that a compiled regexp is some data structure that can be stored once and used again and again. The regexp quote `qr//` does exactly that: `qr/string/` compiles the `string` as a regexp and transforms the result into a form that can be assigned to a variable:

```
$reg = qr/foo+bar?/; # reg contains a compiled regexp
```

Then `$reg` can be used as a regexp:

```
$x = "fooooba";
$x =~ $reg;      # matches, just like /foo+bar?/
$x =~ /$reg/;    # same thing, alternate form
```

`$reg` can also be interpolated into a larger regexp:

```
$x =~ /(abc)?$reg/; # still matches
```

As with the matching operator, the regexp quote can use different delimiters, e.g., `qr!!`, `qr{ }` and `qr~~`. The single quote delimiters `qr' '` prevent any interpolation from taking place.

Pre-compiled regexps are useful for creating dynamic matches that don't need to be recompiled each time they are encountered. Using pre-compiled regexps, `simple_grep` program can be expanded into a program that matches multiple patterns:

```
% cat > multi_grep
#!/usr/bin/perl
# multi_grep - match any of <number> regexps
# usage: multi_grep <number> regexp1 regexp2 ... file1 file2 ...

$number = shift;
$regexps[$_] = shift foreach (0..$number-1);
@compiled = map qr/$_/, @regexps;
while ($line = <>) {
    foreach $pattern (@compiled) {
        if ($line =~ /$pattern/) {
            print $line;
            last; # we matched, so move onto the next line
        }
    }
}
```

```

    }
  }
}
^D

% multi_grep 2 last for multi_grep
$regexp[$_] = shift foreach (0..$number-1);
  foreach $pattern (@compiled) {
    last;
  }
}

```

Storing pre-compiled regexps in an array `@compiled` allows us to simply loop through the regexps without any recompilation, thus gaining flexibility without sacrificing speed.

Embedding comments and modifiers in a regular expression

Starting with this section, we will be discussing Perl's set of **extended patterns**. These are extensions to the traditional regular expression syntax that provide powerful new tools for pattern matching. We have already seen extensions in the form of the minimal matching constructs `??`, `*?`, `+?`, `{n,m}?`, and `{n,}?`. The rest of the extensions below have the form `(?char...)`, where the `char` is a character that determines the type of extension.

The first extension is an embedded comment `(?#text)`. This embeds a comment into the regular expression without affecting its meaning. The comment should not have any closing parentheses in the text. An example is

```
/(?# Match an integer:)[+-]?\d+;/
```

This style of commenting has been largely superseded by the raw, freeform commenting that is allowed with the `//x` modifier.

The modifiers `//i`, `//m`, `//s`, and `//x` can also be embedded in a regexp using `(?i)`, `(?m)`, `(?s)`, and `(?x)`. For instance,

```

/(?i)yes/;  # match 'yes' case insensitively
/yes/i;     # same thing
/(?x)(      # freeform version of an integer regexp
  [+]?     # match an optional sign
  \d+      # match a sequence of digits
)
/x;

```

Embedded modifiers can have two important advantages over the usual modifiers. Embedded modifiers allow a custom set of modifiers to *each* regexp pattern. This is great for matching an array of regexps that must have different modifiers:

```

$pattern[0] = '(?i)doctor';
$pattern[1] = 'Johnson';
...
while (<>) {
  foreach $patt (@pattern) {
    print if /$patt/;
  }
}

```

The second advantage is that embedded modifiers only affect the regexp inside the group the embedded modifier is contained in. So grouping can be used to localize the modifier's effects:

```
/Answer: ((?i)yes)/;  # matches 'Answer: yes', 'Answer: YES', etc.
```

Embedded modifiers can also turn off any modifiers already present by using, e.g., `(?-i)`. Modifiers can also be combined into a single expression, e.g., `(?s-i)` turns on single line mode and turns off case insensitivity.

Non-capturing groupings

We noted in Part 1 that groupings `()` had two distinct functions: 1) group regexp elements together as a single unit, and 2) extract, or capture, substrings that matched the regexp in the grouping. Non-capturing groupings, denoted by `(?:regexp)`, allow the regexp to be treated as a single unit, but don't extract substrings or set matching variables `$1`, etc. Both capturing and non-capturing groupings are allowed to co-exist in the same regexp. Because there is no extraction, non-capturing groupings are faster than capturing groupings. Non-capturing groupings are also handy for choosing exactly which parts of a regexp are to be extracted to matching variables:

```
# match a number, $1-$4 are set, but we only want $1
/([+-]?\ *(\d+(\.\d*)?)|\.\d+)([eE][+-]?\d+)?/;

# match a number faster, only $1 is set
/([+-]?\ *(?:\d+(?:\.\d*)?)|\.\d+)(?:[eE][+-]?\d+)?/;

# match a number, get $1 = whole number, $2 = exponent
/([+-]?\ *(?:\d+(?:\.\d*)?)|\.\d+)(?:[eE]( [+-]?\d+))?/;
```

Non-capturing groupings are also useful for removing nuisance elements gathered from a split operation:

```
$x = '12a34b5';
@num = split /(a|b)/, $x;    # @num = ('12','a','34','b','5')
@num = split /(?:a|b)/, $x;  # @num = ('12','34','5')
```

Non-capturing groupings may also have embedded modifiers: `(?i-m:regexp)` is a non-capturing grouping that matches regexp case insensitively and turns off multi-line mode.

Looking ahead and looking behind

This section concerns the lookahead and lookbehind assertions. First, a little background.

In Perl regular expressions, most regexp elements 'eat up' a certain amount of string when they match. For instance, the regexp element `[abc]` eats up one character of the string when it matches, in the sense that perl moves to the next character position in the string after the match. There are some elements, however, that don't eat up characters (advance the character position) if they match. The examples we have seen so far are the anchors. The anchor `^` matches the beginning of the line, but doesn't eat any characters. Similarly, the word boundary anchor `\b` matches, e.g., if the character to the left is a word character and the character to the right is a non-word character, but it doesn't eat up any characters itself. Anchors are examples of 'zero-width assertions'. Zero-width, because they consume no characters, and assertions, because they test some property of the string. In the context of our walk in the woods analogy to regexp matching, most regexp elements move us along a trail, but anchors have us stop a moment and check our surroundings. If the local environment checks out, we can proceed forward. But if the local environment doesn't satisfy us, we must backtrack.

Checking the environment entails either looking ahead on the trail, looking behind, or both. `^` looks behind, to see that there are no characters before. `$` looks ahead, to see that there are no characters after. `\b` looks both ahead and behind, to see if the characters on either side differ in their 'word'-ness.

The lookahead and lookbehind assertions are generalizations of the anchor concept. Lookahead and lookbehind are zero-width assertions that let us specify which characters we want to test for. The lookahead assertion is denoted by `(?=regexp)` and the lookbehind assertion is denoted by `<(?<=fixed-regexp)`. Some examples are

```
$x = "I catch the housecat 'Tom-cat' with catnip";
$x =~ /cat(=?\s+)/; # matches 'cat' in 'housecat'
@catwords = ($x =~ /(<=?\s)cat\w+/g); # matches,
                                     # $catwords[0] = 'catch'
                                     # $catwords[1] = 'catnip'
$x =~ /\bcat\b/; # matches 'cat' in 'Tom-cat'
$x =~ /(<=?\s)cat(=?\s)/; # doesn't match; no isolated 'cat' in
```

```
# middle of $x
```

Note that the parentheses in `(?=regex)` and `< (?<=regex)` are non-capturing, since these are zero-width assertions. Thus in the second regex, the substrings captured are those of the whole regex itself. Lookahead `(?=regex)` can match arbitrary regexps, but lookbehind `< (?<=fixed-regex)` only works for regexps of fixed width, i.e., a fixed number of characters long. Thus `< (?<=(ab|bc))` is fine, but `< (?<=(ab)*)` is not. The negated versions of the lookahead and lookbehind assertions are denoted by `(?!regex)` and `< (?<=!fixed-regex)` respectively. They evaluate true if the regexps do *not* match:

```
$x = "foobar";
$x =~ /foo(?!bar)/; # doesn't match, 'bar' follows 'foo'
$x =~ /foo(?!baz)/; # matches, 'baz' doesn't follow 'foo'
$x =~ /(?!\\s)foo/; # matches, there is no \\s before 'foo'
```

Using independent subexpressions to prevent backtracking

The last few extended patterns in this tutorial are experimental as of 5.6.0. Play with them, use them in some code, but don't rely on them just yet for production code.

Independent subexpressions are regular expressions, in the context of a larger regular expression, that function independently of the larger regular expression. That is, they consume as much or as little of the string as they wish without regard for the ability of the larger regex to match. Independent subexpressions are represented by `< (?regex)`. We can illustrate their behavior by first considering an ordinary regex:

```
$x = "ab";
$x =~ /a*ab/; # matches
```

This obviously matches, but in the process of matching, the subexpression `a*` first grabbed the `a`. Doing so, however, wouldn't allow the whole regex to match, so after backtracking, `a*` eventually gave back the `a` and matched the empty string. Here, what `a*` matched was *dependent* on what the rest of the regex matched.

Contrast that with an independent subexpression:

```
$x =~ /(?!>a*)ab/; # doesn't match!
```

The independent subexpression `< (?a*)` doesn't care about the rest of the regex, so it sees an `a` and grabs it. Then the rest of the regex `ab` cannot match. Because `< (?a*)` is independent, there is no backtracking and the independent subexpression does not give up its `a`. Thus the match of the regex as a whole fails. A similar behavior occurs with completely independent regexps:

```
$x = "ab";
$x =~ /a*/g; # matches, eats an 'a'
$x =~ /\Gab/g; # doesn't match, no 'a' available
```

Here `//g` and `\G` create a 'tag team' handoff of the string from one regex to the other. Regexps with an independent subexpression are much like this, with a handoff of the string to the independent subexpression, and a handoff of the string back to the enclosing regex.

The ability of an independent subexpression to prevent backtracking can be quite useful. Suppose we want to match a non-empty string enclosed in parentheses up to two levels deep. Then the following regex matches:

```
$x = "abc(de(fg)h)"; # unbalanced parentheses
$x =~ /\( ( [^()]+ | \( [^()]* \) )+ \| \)/x;
```

The regex matches an open parenthesis, one or more copies of an alternation, and a close parenthesis. The alternation is two-way, with the first alternative `[^()]+` matching a substring with no parentheses and the second alternative `\([^()]* \)` matching a substring delimited by parentheses. The problem with this regex is that it is pathological: it has nested indeterminate quantifiers of the form `(a+|b)+`. We discussed in Part 1 how nested quantifiers like this could take an exponentially long time to execute if there was no

match possible. To prevent the exponential blowup, we need to prevent useless backtracking at some point. This can be done by enclosing the inner quantifier as an independent subexpression:

```
$x =~ /\( ( (?>[^\()]+) | \( [^\()]* \) )+ \)/x;
```

Here, `< (?>[^\()]+)` breaks the degeneracy of string partitioning by gobbling up as much of the string as possible and keeping it. Then match failures fail much more quickly.

Conditional expressions

A **conditional expression** is a form of if–then–else statement that allows one to choose which patterns are to be matched, based on some condition. There are two types of conditional expression:

`(?(condition)yes-regexp)` and `(?(condition)yes-regexp|no-regexp)`.

`(?(condition)yes-regexp)` is like an `'if () {}'` statement in Perl. If the condition is true, the `yes-regexp` will be matched. If the condition is false, the `yes-regexp` will be skipped and perl will move onto the next regexp element. The second form is like an `'if () {} else {}'` statement in Perl. If the condition is true, the `yes-regexp` will be matched, otherwise the `no-regexp` will be matched.

The condition can have two forms. The first form is simply an integer in parentheses `(integer)`. It is true if the corresponding backreference `\integer` matched earlier in the regexp. The second form is a bare zero width assertion `(?...)`, either a lookahead, a lookbehind, or a code assertion (discussed in the next section).

The integer form of the condition allows us to choose, with more flexibility, what to match based on what matched earlier in the regexp. This searches for words of the form `"xx"` or `"xyyx"`:

```
% simple_grep '^(\w+)(\w+)?(?(2)\2\1|\1)$' /usr/dict/words
beriberi
coco
couscous
deed
...
toot
toto
tutu
```

The lookbehind condition allows, along with backreferences, an earlier part of the match to influence a later part of the match. For instance,

```
/ [ATGC] + (?(?<=AA)G|C) $/;
```

matches a DNA sequence such that it either ends in AAG, or some other base pair combination and C. Note that the form is `< (?(?<=AA)G|C)` and not `< (?(?<=AA))G|C`; for the lookahead, lookbehind or code assertions, the parentheses around the conditional are not needed.

A bit of magic: executing Perl code in a regular expression

Normally, regexps are a part of Perl expressions. **Code evaluation** expressions turn that around by allowing arbitrary Perl code to be a part of a regexp. A code evaluation expression is denoted `(?{code})`, with code a string of Perl statements.

Code expressions are zero-width assertions, and the value they return depends on their environment. There are two possibilities: either the code expression is used as a conditional in a conditional expression `(?(condition)...)` , or it is not. If the code expression is a conditional, the code is evaluated and the result (i.e., the result of the last statement) is used to determine truth or falsehood. If the code expression is not used as a conditional, the assertion always evaluates true and the result is put into the special variable `$_R`. The variable `$_R` can then be used in code expressions later in the regexp. Here are some silly examples:

```
$x = "abcdef";
```

```
$x =~ /abc(?:print "Hi Mom!";)def/; # matches,
                                     # prints 'Hi Mom!'
$x =~ /aaa(?:print "Hi Mom!";)def/; # doesn't match,
                                     # no 'Hi Mom!'
```

Pay careful attention to the next example:

```
$x =~ /abc(?:print "Hi Mom!";)ddd/; # doesn't match,
                                     # no 'Hi Mom!'
                                     # but why not?
```

At first glance, you'd think that it shouldn't print, because obviously the ddd isn't going to match the target string. But look at this example:

```
$x =~ /abc(?:print "Hi Mom!";)[d]dd/; # doesn't match,
                                     # but does print
```

Hmm. What happened here? If you've been following along, you know that the above pattern should be effectively the same as the last one — enclosing the d in a character class isn't going to change what it matches. So why does the first not print while the second one does?

The answer lies in the optimizations the REx engine makes. In the first case, all the engine sees are plain old characters (aside from the `{}` construct). It's smart enough to realize that the string 'ddd' doesn't occur in our target string before actually running the pattern through. But in the second case, we've tricked it into thinking that our pattern is more complicated than it is. It takes a look, sees our character class, and decides that it will have to actually run the pattern to determine whether or not it matches, and in the process of running it hits the print statement before it discovers that we don't have a match.

To take a closer look at how the engine does optimizations, see the section *"Pragmas and debugging"* below.

More fun with `{}` :

```
$x =~ /(?:print "Hi Mom!";)/;      # matches,
                                   # prints 'Hi Mom!'
$x =~ /(?:{$c = 1;})(?:print "$c";)/; # matches,
                                   # prints '1'
$x =~ /(?:{$c = 1;})(?:print "$^R";)/; # matches,
                                   # prints '1'
```

The bit of magic mentioned in the section title occurs when the regexp backtracks in the process of searching for a match. If the regexp backtracks over a code expression and if the variables used within are localized using `local`, the changes in the variables produced by the code expression are undone! Thus, if we wanted to count how many times a character got matched inside a group, we could use, e.g.,

```
$x = "aaaa";
$count = 0; # initialize 'a' count
$c = "bob"; # test if $c gets clobbered
$x =~ /(?:{local $c = 0;}
    ( a # match 'a'
      (?:{local $c = $c + 1;} # increment count
    )* # do this any number of times,
    aa # but match 'aa' at the end
    (?:{$count = $c;} # copy local $c var into $count
  )/x;
print "'a' count is $count, \"$c\" variable is '$c'\n";
```

This prints

```
'a' count is 2, $c variable is 'bob'
```

If we replace the `(?{local $c = $c + 1;})` with `(?{$c = $c + 1;})`, the variable changes are *not* undone during backtracking, and we get

```
'a' count is 4, $c variable is 'bob'
```

Note that only localized variable changes are undone. Other side effects of code expression execution are permanent. Thus

```
$x = "aaaa";
$x =~ /(a(?{print "Yow\n";})) *aa/;
```

produces

```
Yow
Yow
Yow
Yow
```

The result `$^R` is automatically localized, so that it will behave properly in the presence of backtracking.

This example uses a code expression in a conditional to match the article ‘the’ in either English or German:

```
$lang = 'DE'; # use German
...
$text = "das";
print "matched\n"
    if $text =~ /(?(?{
        $lang eq 'EN'; # is the language English?
    })
    the |                # if so, then match 'the'
    (die|das|der)        # else, match 'die|das|der'
    )
    /xi;
```

Note that the syntax here is `(?(?{...})yes-regexp|no-regexp)`, not `(?(?(?{...}))yes-regexp|no-regexp)`. In other words, in the case of a code expression, we don’t need the extra parentheses around the conditional.

If you try to use code expressions with interpolating variables, perl may surprise you:

```
$bar = 5;
$pat = '(?{ 1 })';
/foo(?{ $bar })bar/; # compiles ok, $bar not interpolated
/foo(?{ 1 })$bar/;   # compile error!
/foo${pat}bar/;      # compile error!

$pat = qr/(?{ $foo = 1 })/; # precompile code regexp
/foo${pat}bar/;           # compiles ok
```

If a regexp has (1) code expressions and interpolating variables, or (2) a variable that interpolates a code expression, perl treats the regexp as an error. If the code expression is precompiled into a variable, however, interpolating is ok. The question is, why is this an error?

The reason is that variable interpolation and code expressions together pose a security risk. The combination is dangerous because many programmers who write search engines often take user input and plug it directly into a regexp:

```
$regexp = <>; # read user-supplied regexp
schomp $regexp; # get rid of possible newline
$text =~ /$regexp/; # search $text for the $regexp
```

If the `$regexp` variable contains a code expression, the user could then execute arbitrary Perl code. For

instance, some joker could search for `system('rm -rf *');` to erase your files. In this sense, the combination of interpolation and code expressions **taints** your regexp. So by default, using both interpolation and code expressions in the same regexp is not allowed. If you're not concerned about malicious users, it is possible to bypass this security check by invoking `use re 'eval'`:

```
use re 'eval';          # throw caution out the door
$bar = 5;
$pat = '(?{ 1 })';
/foo(?{ 1 })$bar/;      # compiles ok
/foo${pat}bar/;         # compiles ok
```

Another form of code expression is the **pattern code expression**. The pattern code expression is like a regular code expression, except that the result of the code evaluation is treated as a regular expression and matched immediately. A simple example is

```
$length = 5;
$char = 'a';
$x = 'aaaaabb';
$x =~ /(??{$char x $length})/x; # matches, there are 5 of 'a'
```

This final example contains both ordinary and pattern code expressions. It detects if a binary string 1101010010001... has a Fibonacci spacing 0,1,1,2,3,5,... of the 1's:

```
$s0 = 0; $s1 = 1; # initial conditions
$x = "1101010010001000001";
print "It is a Fibonacci sequence\n"
    if $x =~ /^1
        (
            (??{'0' x $s0}) # match $s0 of '0'
            1               # and then a '1'
            (?{
                $largest = $s0; # largest seq so far
                $s2 = $s1 + $s0; # compute next term
                $s0 = $s1;      # in Fibonacci sequence
                $s1 = $s2;
            })
        )+ # repeat as needed
    $     # that is all there is
    /x;

print "Largest sequence matched was $largest\n";
```

This prints

```
It is a Fibonacci sequence
Largest sequence matched was 5
```

Ha! Try that with your garden variety regexp package...

Note that the variables `$s0` and `$s1` are not substituted when the regexp is compiled, as happens for ordinary variables outside a code expression. Rather, the code expressions are evaluated when perl encounters them during the search for a match.

The regexp without the `//x` modifier is

```
/^1((??{'0'x$s0})1(?{$largest=$s0;$s2=$s1+$s0;$s0=$s1;$s1=$s2;}))+$/;
```

and is a great start on an Obfuscated Perl entry :-) When working with code and conditional expressions, the extended form of regexps is almost necessary in creating and debugging regexps.

Pragmas and debugging

Speaking of debugging, there are several pragmas available to control and debug regexps in Perl. We have already encountered one pragma in the previous section, `use re 'eval'`, that allows variable interpolation and code expressions to coexist in a regexp. The other pragmas are

```
use re 'taint';
$tainted = <>;
@parts = ($tainted =~ /(\w+)\s+(\w+)/; # @parts is now tainted
```

The `taint` pragma causes any substrings from a match with a tainted variable to be tainted as well. This is not normally the case, as regexps are often used to extract the safe bits from a tainted variable. Use `taint` when you are not extracting safe bits, but are performing some other processing. Both `taint` and `eval` pragmas are lexically scoped, which means they are in effect only until the end of the block enclosing the pragmas.

```
use re 'debug';
/^(.*)$/s;          # output debugging info

use re 'debugcolor';
/^(.*)$/s;          # output debugging info in living color
```

The global `debug` and `debugcolor` pragmas allow one to get detailed debugging info about regexp compilation and execution. `debugcolor` is the same as `debug`, except the debugging information is displayed in color on terminals that can display termcap color sequences. Here is example output:

```
% perl -e 'use re "debug"; "abc" =~ /a*b+c/;'
Compiling REx 'a*b+c'
size 9 first at 1
  1: STAR(4)
  2:   EXACT <a>(0)
  4: PLUS(7)
  5:   EXACT <b>(0)
  7: EXACT <c>(9)
  9: END(0)
floating 'bc' at 0..2147483647 (checking floating) minlen 2
Guessing start of match, REx 'a*b+c' against 'abc'...
Found floating substr 'bc' at offset 1...
Guessed: match at offset 0
Matching REx 'a*b+c' against 'abc'
  Setting an EVAL scope, savestack=3
    0 <> <abc>          |  1:  STAR
                        |    EXACT <a> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
    1 <a> <bc>          |  4:  PLUS
                        |    EXACT <b> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
    2 <ab> <c>          |  7:    EXACT <c>
    3 <abc> <>          |  9:    END
Match successful!
Freeing REx: 'a*b+c'
```

If you have gotten this far into the tutorial, you can probably guess what the different parts of the debugging output tell you. The first part

```
Compiling REx 'a*b+c'
size 9 first at 1
  1: STAR(4)
  2:   EXACT <a>(0)
```

```

4: PLUS (7)
5:   EXACT <b>(0)
7: EXACT <c>(9)
9: END (0)

```

describes the compilation stage. STAR(4) means that there is a starred object, in this case 'a', and if it matches, goto line 4, i.e., PLUS(7). The middle lines describe some heuristics and optimizations performed before a match:

```

floating 'bc' at 0..2147483647 (checking floating) minlen 2
Guessing start of match, REx 'a*b+c' against 'abc'...
Found floating substr 'bc' at offset 1...
Guessed: match at offset 0

```

Then the match is executed and the remaining lines describe the process:

```

Matching REx 'a*b+c' against 'abc'
Setting an EVAL scope, savestack=3
0 <> <abc>          | 1:  STAR
                     EXACT <a> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
1 <a> <bc>          | 4:  PLUS
                     EXACT <b> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
2 <ab> <c>          | 7:  EXACT <c>
3 <abc> <>          | 9:  END
Match successful!
Freeing REx: 'a*b+c'

```

Each step is of the form < n <x>y , with < <x> the part of the string matched and < <y> the part not yet matched. The < | 1: STAR says that perl is at line number 1 in the compilation list above. See [Debugging regular expressions in perldebguts](#) for much more detail.

An alternative method of debugging regexps is to embed print statements within the regexp. This provides a blow-by-blow account of the backtracking in an alternation:

```

"that this" =~ m@(?{print "Start at position ", pos, "\n";})
                t(?{print "t1\n";})
                h(?{print "h1\n";})
                i(?{print "i1\n";})
                s(?{print "s1\n";})
                |
                t(?{print "t2\n";})
                h(?{print "h2\n";})
                a(?{print "a2\n";})
                t(?{print "t2\n";})
                (?{print "Done at position ", pos, "\n";})
                @x;

```

prints

```

Start at position 0
t1
h1
t2
h2
a2
t2
Done at position 4

```


BUGS

Code expressions, conditional expressions, and independent expressions are **experimental**. Don't use them in production code. Yet.

SEE ALSO

This is just a tutorial. For the full story on perl regular expressions, see the [perlre](#) regular expressions reference page.

For more information on the matching `m//` and substitution `s///` operators, see [Regexp Quote-Like Operators in perlop](#). For information on the `split` operation, see [split](#).

For an excellent all-around resource on the care and feeding of regular expressions, see the book *Mastering Regular Expressions* by Jeffrey Friedl (published by O'Reilly, ISBN 1556592-257-3).

AUTHOR AND COPYRIGHT

Copyright (c) 2000 Mark Kvale All rights reserved.

This document may be distributed under the same terms as Perl itself.

Acknowledgments

The inspiration for the stop codon DNA example came from the ZIP code example in chapter 7 of *Mastering Regular Expressions*.

The author would like to thank Jeff Pinyan, Andrew Johnson, Peter Haworth, Ronald J Kimball, and Joe Smith for all their helpful comments.

NAME

perlrun – how to execute the Perl interpreter

SYNOPSIS

```
perl [ -CsTtuUWX ]
      [ -hv ] [ -V[:configvar] ]
      [ -cw ] [ -d[:debugger] ] [ -D[number/list] ]
      [ -pna ] [ -Fpattern ] [ -I[octal] ] [ -O[octal] ]
      [ -Idir ] [ -m[-]module ] [ -M[-]'module...' ]
      [ -P ]
      [ -S ]
      [ -x[dir] ]
      [ -i[extension] ]
      [ -e 'command' ] [ — ] [ programfile ] [ argument ]...
```

DESCRIPTION

The normal way to run a Perl program is by making it directly executable, or else by passing the name of the source file as an argument on the command line. (An interactive Perl environment is also possible—see [perldebug](#) for details on how to do that.) Upon startup, Perl looks for your program in one of the following places:

1. Specified line by line via **-e** switches on the command line.
2. Contained in the file specified by the first filename on the command line. (Note that systems supporting the **#!** notation invoke interpreters this way. See [Location of Perl](#).)
3. Passed in implicitly via standard input. This works only if there are no filename arguments—to pass arguments to a STDIN-read program you must explicitly specify a **"-"** for the program name.

With methods 2 and 3, Perl starts parsing the input file from the beginning, unless you've specified a **-x** switch, in which case it scans for the first line starting with **#!** and containing the word "perl", and starts there instead. This is useful for running a program embedded in a larger message. (In this case you would indicate the end of the program using the `__END__` token.)

The **#!** line is always examined for switches as the line is being parsed. Thus, if you're on a machine that allows only one argument with the **#!** line, or worse, doesn't even recognize the **#!** line, you still can get consistent switch behavior regardless of how Perl was invoked, even if **-x** was used to find the beginning of the program.

Because historically some operating systems silently chopped off kernel interpretation of the **#!** line after 32 characters, some switches may be passed in on the command line, and some may not; you could even get a **"-"** without its letter, if you're not careful. You probably want to make sure that all your switches fall either before or after that 32-character boundary. Most switches don't actually care if they're processed redundantly, but getting a **"-"** instead of a complete switch could cause Perl to try to execute standard input instead of your program. And a partial **-I** switch could also cause odd results.

Some switches do care if they are processed twice, for instance combinations of **-I** and **-O**. Either put all the switches after the 32-character boundary (if applicable), or replace the use of **-Odigits** by `BEGIN{ $/ = "\0digits"; }`.

Parsing of the **#!** switches starts wherever "perl" is mentioned in the line. The sequences **"-*"** and **"- "** are specifically ignored so that you could, if you were so inclined, say

```
#!/bin/sh -- # -* perl -* -p
eval 'exec perl -wS $0 ${1+"$@"}'
    if $running_under_some_shell;
```

to let Perl see the **-p** switch.

A similar trick involves the **env** program, if you have it.

```
#!/usr/bin/env perl
```

The examples above use a relative path to the perl interpreter, getting whatever version is first in the user's path. If you want a specific version of Perl, say, perl5.005_57, you should place that directly in the #! line's path.

If the #! line does not contain the word "perl", the program named after the #! is executed instead of the Perl interpreter. This is slightly bizarre, but it helps people on machines that don't do #!, because they can tell a program that their SHELL is */usr/bin/perl*, and Perl will then dispatch the program to the correct interpreter for them.

After locating your program, Perl compiles the entire program to an internal form. If there are any compilation errors, execution of the program is not attempted. (This is unlike the typical shell script, which might run part-way through before finding a syntax error.)

If the program is syntactically correct, it is executed. If the program runs off the end without hitting an `exit()` or `die()` operator, an implicit `exit(0)` is provided to indicate successful completion.

#! and quoting on non-Unix systems

Unix's #! technique can be simulated on other systems:

OS/2

Put

```
extproc perl -S -your_switches
```

as the first line in *.cmd file (**-S** due to a bug in cmd.exe's 'extproc' handling).

MS-DOS

Create a batch file to run your program, and codify it in `ALTERNATIVE_SHEBANG` (see the *dosish.h* file in the source distribution for more information).

Win95/NT

The Win95/NT installation, when using the ActiveState installer for Perl, will modify the Registry to associate the *.pl* extension with the perl interpreter. If you install Perl by other means (including building from the sources), you may have to modify the Registry yourself. Note that this means you can no longer tell the difference between an executable Perl program and a Perl library file.

Macintosh

A Macintosh perl program will have the appropriate Creator and Type, so that double-clicking them will invoke the perl application.

VMS

Put

```
$ perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8' !
$ exit++ + ++$status != 0 and $exit = $status = undef;
```

at the top of your program, where **-mysw** are any command line switches you want to pass to Perl. You can now invoke the program directly, by saying `perl program`, or as a DCL procedure, by saying `@program` (or implicitly via **DCL\$PATH** by just using the name of the program).

This incantation is a bit much to remember, but Perl will display it for you if you say `perl -V:startperl`.

Command-interpreters on non-Unix systems have rather different ideas on quoting than Unix shells. You'll need to learn the special characters in your command-interpreter (*, \ and " are common) and how to protect whitespace and these characters to run one-liners (see **-e** below).

On some systems, you may have to change single-quotes to double ones, which you must *not* do on Unix or

Plan 9 systems. You might also have to change a single % to a %%.

For example:

```
# Unix
perl -e 'print "Hello world\n"'

# MS-DOS, etc.
perl -e "print \"Hello world\n\\\"\"

# Macintosh
print "Hello world\n"
(then Run "Myscript" or Shift-Command-R)

# VMS
perl -e "print \"Hello world\n\""
```

The problem is that none of this is reliable: it depends on the command and it is entirely possible neither works. If **4DOS** were the command shell, this would probably work better:

```
perl -e "print <Ctrl-x>\"Hello world\n<Ctrl-x>\""
```

CMD.EXE in Windows NT slipped a lot of standard Unix functionality in when nobody was looking, but just try to find documentation for its quoting rules.

Under the Macintosh, it depends which environment you are using. The MacPerl shell, or MPW, is much like Unix shells in its support for several quoting variants, except that it makes free use of the Macintosh's non-ASCII characters as control characters.

There is no general solution to all of this. It's just a mess.

Location of Perl

It may seem obvious to say, but Perl is useful only when users can easily find it. When possible, it's good for both `/usr/bin/perl` and `/usr/local/bin/perl` to be symlinks to the actual binary. If that can't be done, system administrators are strongly encouraged to put (symlinks to) perl and its accompanying utilities into a directory typically found along a user's PATH, or in some other obvious and convenient place.

In this documentation, `#!/usr/bin/perl` on the first line of the program will stand in for whatever method works on your system. You are advised to use a specific path if you care about a specific version.

```
#!/usr/local/bin/perl5.00554
```

or if you just want to be running at least version, place a statement like this at the top of your program:

```
use 5.005_54;
```

Command Switches

As with all standard commands, a single-character switch may be clustered with the following switch, if any.

```
#!/usr/bin/perl -spi.orig # same as -s -p -i.orig
```

Switches include:

-O[*digits*]

specifies the input record separator (`$/`) as an octal number. If there are no digits, the null character is the separator. Other switches may precede or follow the digits. For example, if you have a version of **find** which can print filenames terminated by the null character, you can say this:

```
find . -name '*.orig' -print0 | perl -n0e unlink
```

The special value 00 will cause Perl to slurp files in paragraph mode. The value 0777 will cause Perl to slurp files whole because there is no legal character with that value.

- a** turns on autosplit mode when used with a **-n** or **-p**. An implicit split command to the `@F` array is done as the first thing inside the implicit while loop produced by the **-n** or **-p**.

```
perl -ane 'print pop(@F), "\n";'
```

is equivalent to

```
while (<>) {
    @F = split(' ');
    print pop(@F), "\n";
}
```

An alternate delimiter may be specified using **-F**.

- C** enables Perl to use the native wide character APIs on the target system. The magic variable `${^WIDE_SYSTEM_CALLS}` reflects the state of this switch. See [\\${^WIDE_SYSTEM_CALLS} in perlvar](#).

This feature is currently only implemented on the Win32 platform.

- c** causes Perl to check the syntax of the program and then exit without executing it. Actually, it *will* execute BEGIN, CHECK, and use blocks, because these are considered as occurring outside the execution of your program. INIT and END blocks, however, will be skipped.

- d** runs the program under the Perl debugger. See [perldebug](#).

-d:foo[=bar,baz]

runs the program under the control of a debugging, profiling, or tracing module installed as `Devel::foo`. E.g., **-d:DProf** executes the program using the `Devel::DProf` profiler. As with the **-M** flag, options may be passed to the `Devel::foo` package where they will be received and interpreted by the `Devel::foo::import` routine. The comma-separated list of options must follow a `=` character. See [perldebug](#).

-Dletters

-Dnumber

sets debugging flags. To watch how it executes your program, use **-Dtls**. (This works only if debugging is compiled into your Perl.) Another nice value is **-Dx**, which lists your compiled syntax tree. And **-Dr** displays compiled regular expressions; the format of the output is explained in [perldebugs](#).

As an alternative, specify a number instead of list of letters (e.g., **-D14** is equivalent to **-Dtls**):

```
1  p  Tokenizing and parsing
2  s  Stack snapshots
4  l  Context (loop) stack processing
8  t  Trace execution
16 o  Method and overloading resolution
32 c  String/numeric conversions
64 P  Print profiling info, preprocessor command for -P, source file info
128 m Memory allocation
256 f  Format processing
512 r  Regular expression parsing and execution
1024 x Syntax tree dump
2048 u Tainting checks
4096 L Memory leaks (needs -DLEAKTEST when compiling Perl)
8192 H Hash dump -- usurps values()
16384 X Scratchpad allocation
32768 D Cleaning up
65536 S Thread synchronization
131072 T Tokenising
```

```
262144 R Include reference counts of dumped variables (eg when using -Ds)
524288 J Do not s,t,P-debug (Jump over) opcodes within package DB
```

All these flags require **-DDEBUGGING** when you compile the Perl executable (but see [Devel::Peek](#), [re](#) which may change this). See the **INSTALL** file in the Perl source distribution for how to do this. This flag is automatically set if you include **-g** option when Configure asks you about optimizer/debugger flags.

If you're just trying to get a print out of each line of Perl code as it executes, the way that `sh -x` provides for shell scripts, you can't use Perl's **-D** switch. Instead do this

```
# If you have "env" utility
env=PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# Bourne shell syntax
$ PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# csh syntax
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS program)
```

See [perldebug](#) for details and variations.

-e *commandline*

may be used to enter one line of program. If **-e** is given, Perl will not look for a filename in the argument list. Multiple **-e** commands may be given to build up a multi-line script. Make sure to use semicolons where you would in a normal program.

-F *pattern*

specifies the pattern to split on if **-a** is also in effect. The pattern may be surrounded by `//`, `"`, or `'`, otherwise it will be put in single quotes.

-h prints a summary of the options.

-i[*extension*]

specifies that files processed by the `<>` construct are to be edited in-place. It does this by renaming the input file, opening the output file by the original name, and selecting that output file as the default for `print()` statements. The extension, if supplied, is used to modify the name of the old file to make a backup copy, following these rules:

If no extension is supplied, no backup is made and the current file is overwritten.

If the extension doesn't contain a `*`, then it is appended to the end of the current filename as a suffix. If the extension does contain one or more `*` characters, then each `*` is replaced with the current filename. In Perl terms, you could think of this as:

```
($backup = $extension) =~ s/\*/$file_name/g;
```

This allows you to add a prefix to the backup file, instead of (or in addition to) a suffix:

```
$ perl -pi 'orig_*' -e 's/bar/baz/' fileA # backup to 'orig_fileA'
```

Or even to place backup copies of the original files into another directory (provided the directory already exists):

```
$ perl -pi 'old/*.orig' -e 's/bar/baz/' fileA # backup to 'old/fileA.orig'
```

These sets of one-liners are equivalent:

```
$ perl -pi -e 's/bar/baz/' fileA # overwrite current file
$ perl -pi '*' -e 's/bar/baz/' fileA # overwrite current file

$ perl -pi '.orig' -e 's/bar/baz/' fileA # backup to 'fileA.orig'
$ perl -pi '*.orig' -e 's/bar/baz/' fileA # backup to 'fileA.orig'
```

From the shell, saying

```
$ perl -p -i.orig -e "s/foo/bar/; ... "
```

is the same as using the program:

```
#!/usr/bin/perl -pi.orig
s/foo/bar/;
```

which is equivalent to

```
#!/usr/bin/perl
$extension = '.orig';
LINE: while (<>) {
    if ($ARGV ne $oldargv) {
        if ($extension !~ /\*/) {
            $backup = $ARGV . $extension;
        }
        else {
            ($backup = $extension) =~ s/\*/$ARGV/g;
        }
        rename($ARGV, $backup);
        open(ARGVOUT, ">$ARGV");
        select(ARGVOUT);
        $oldargv = $ARGV;
    }
    s/foo/bar/;
}
continue {
    print; # this prints to original filename
}
select(STDOUT);
```

except that the `-i` form doesn't need to compare `$ARGV` to `$oldargv` to know when the filename has changed. It does, however, use `ARGVOUT` for the selected filehandle. Note that `STDOUT` is restored as the default output filehandle after the loop.

As shown above, Perl creates the backup file whether or not any output is actually changed. So this is just a fancy way to copy files:

```
$ perl -p -i '/some/file/path/*' -e 1 file1 file2 file3...
```

or

```
$ perl -p -i '.orig' -e 1 file1 file2 file3...
```

You can use `eof` without parentheses to locate the end of each input file, in case you want to append to each file, or reset line numbering (see example in [eof](#)).

If, for a given file, Perl is unable to create the backup file as specified in the extension then it will skip that file and continue on with the next one (if it exists).

For a discussion of issues surrounding file permissions and `-i`, see

[Why does Perl let me delete read-only files? Why does -i clobber protected files? Isn't this a bug in Perl?](#)

You cannot use `-i` to create directories or to strip extensions from files.

Perl does not expand `~` in filenames, which is good, since some folks use it for their backup files:

```
$ perl -pi~ -e 's/foo/bar/' file1 file2 file3...
```

Finally, the `-i` switch does not impede execution when no files are given on the command line. In this case, no backup is made (the original file cannot, of course, be determined) and processing

proceeds from STDIN to STDOUT as might be expected.

-Idirectory

Directories specified by **-I** are prepended to the search path for modules (@INC), and also tells the C preprocessor where to search for include files. The C preprocessor is invoked with **-P**; by default it searches /usr/include and /usr/lib/perl.

-l[*octnum*]

enables automatic line-ending processing. It has two separate effects. First, it automatically chomps `$/` (the input record separator) when used with **-n** or **-p**. Second, it assigns `$\` (the output record separator) to have the value of *octnum* so that any print statements will have that separator added back on. If *octnum* is omitted, sets `$\` to the current value of `$/`. For instance, to trim lines to 80 columns:

```
perl -lpe 'substr($_, 80) = ""'
```

Note that the assignment `$\ = $/` is done when the switch is processed, so the input record separator can be different than the output record separator if the **-l** switch is followed by a **-0** switch:

```
gnumfind / -print0 | perl -ln0e 'print "found $_" if -p'
```

This sets `$\` to newline and then sets `$/` to the null character.

-m[-]*module*

-M[-]*module*

-M[-]'*module ...*'

-[mM][-]*module=arg[,arg]...*

-mmodule executes `use module ();` before executing your program.

-Mmodule executes `use module ;` before executing your program. You can use quotes to add extra code after the module name, e.g., `'-Mmodule qw(foo bar)'`.

If the first character after the **-M** or **-m** is a dash (**-**) then the 'use' is replaced with 'no'.

A little builtin syntactic sugar means you can also say **-mmodule=foo,bar** or **-Mmodule=foo,bar** as a shortcut for `'-Mmodule qw(foo bar)'`. This avoids the need to use quotes when importing symbols. The actual code generated by **-Mmodule=foo,bar** is `use module split(/,/,q{foo,bar})`. Note that the `=` form removes the distinction between **-m** and **-M**.

-n causes Perl to assume the following loop around your program, which makes it iterate over filename arguments somewhat like **sed -n** or **awk**:

```
LINE:
    while (<>) {
        ...                # your program goes here
    }
```

Note that the lines are not printed by default. See **-p** to have lines printed. If a file named by an argument cannot be opened for some reason, Perl warns you about it and moves on to the next file.

Here is an efficient way to delete all files older than a week:

```
find . -mtime +7 -print | perl -nle unlink
```

This is faster than using the **-exec** switch of **find** because you don't have to start a process on every filename found. It does suffer from the bug of mishandling newlines in pathnames, which you can fix if you follow the example under **-0**.

BEGIN and END blocks may be used to capture control before or after the implicit program loop, just as in **awk**.

- p** causes Perl to assume the following loop around your program, which makes it iterate over filename arguments somewhat like **sed**:

```

LINE:
    while (<>) {
        ...                # your program goes here
    } continue {
        print or die "-p destination: $!\n";
    }

```

If a file named by an argument cannot be opened for some reason, Perl warns you about it, and moves on to the next file. Note that the lines are printed automatically. An error occurring during printing is treated as fatal. To suppress printing use the **-n** switch. A **-p** overrides a **-n** switch.

BEGIN and END blocks may be used to capture control before or after the implicit loop, just as in **awk**.

- P** **NOTE: Use of -P is strongly discouraged because of its inherent problems, including poor portability.**

This option causes your program to be run through the C preprocessor before compilation by Perl. Because both comments and **cpp** directives begin with the **#** character, you should avoid starting comments with any words recognized by the C preprocessor such as "if", "else", or "define".

If you're considering using **-P**, you might also want to look at the `Filter::cpp` module from CPAN.

The problems of **-P** include, but are not limited to:

- The **#!** line is stripped, so any switches there don't apply.
- A **-P** on a **#!** line doesn't work.
- **All** lines that begin with (whitespace and) a **#** but do not look like **cpp** commands, are stripped, including anything inside Perl strings, regular expressions, and here-docs .
- In some platforms the C preprocessor knows too much: it knows about the C++ **-style** until-end-of-line comments starting with **"/"**. This will cause problems with common Perl constructs like

```
s/foo//;
```

because after **-P** this will become illegal code

```
s/foo
```

The workaround is to use some other quoting separator than **"/"**, like for example **"! "**:

```
s!foo!;
```

- It requires not only a working C preprocessor but also a working **sed**. If not on UNIX, you are probably out of luck on this.
 - Script line numbers are not preserved.
 - The **-x** does not work with **-P**.
- s** enables rudimentary switch parsing for switches on the command line after the program name but before any filename arguments (or before an argument of **—**). This means you can have switches with two leading dashes (**—help**). Any switch found there is removed from **@ARGV** and sets the corresponding variable in the Perl program. The following program prints "1" if the program is invoked with a **-xyz** switch, and "abc" if it is invoked with **-xyz=abc**.

```
#!/usr/bin/perl -s
if ($xyz) { print "$xyz\n" }
```

Do note that **—help** creates the variable `${-help}`, which is not compliant with `strict refs`.

- S** makes Perl use the `PATH` environment variable to search for the program (unless the name of the program contains directory separators).

On some platforms, this also makes Perl append suffixes to the filename while searching for it. For example, on Win32 platforms, the ".bat" and ".cmd" suffixes are appended if a lookup for the original name fails, and if the name does not already end in one of those suffixes. If your Perl was compiled with `DEBUGGING` turned on, using the `—Dp` switch to Perl shows how the search progresses.

Typically this is used to emulate `#!` startup on platforms that don't support `#!`. This example works on many platforms that have a shell compatible with Bourne shell:

```
#!/usr/bin/perl
eval 'exec /usr/bin/perl -wS $0 ${1+"$@"}'
if $running_under_some_shell;
```

The system ignores the first line and feeds the program to `/bin/sh`, which proceeds to try to execute the Perl program as a shell script. The shell executes the second line as a normal shell command, and thus starts up the Perl interpreter. On some systems `$0` doesn't always contain the full pathname, so the `—S` tells Perl to search for the program if necessary. After Perl locates the program, it parses the lines and ignores them because the variable `$running_under_some_shell` is never true. If the program will be interpreted by `csh`, you will need to replace `${1+"$@"}` with `$*`, even though that doesn't understand embedded spaces (and such) in the argument list. To start up `sh` rather than `csh`, some systems may have to replace the `#!` line with a line containing just a colon, which will be politely ignored by Perl. Other systems can't control that, and need a totally devious construct that will work under any of `csh`, `sh`, or Perl, such as the following:

```
eval '(exit $?0)' && eval 'exec perl -wS $0 ${1+"$@"}'
& eval 'exec /usr/bin/perl -wS $0 $argv:q'
if $running_under_some_shell;
```

If the filename supplied contains directory separators (i.e., is an absolute or relative pathname), and if that file is not found, platforms that append file extensions will do so and try to look for the file with those extensions added, one by one.

On DOS-like platforms, if the program does not contain directory separators, it will first be searched for in the current directory before being searched for on the `PATH`. On Unix platforms, the program will be searched for strictly on the `PATH`.

- t** Like `—T`, but taint checks will issue warnings rather than fatal errors. These warnings can be controlled normally with `no warnings qw(taint)`.

NOTE: this is not a substitute for `—T`. This is meant only to be used as a temporary development aid while securing legacy code: for real production code and for new secure code written from scratch always use the real `—T`.

- T** forces "taint" checks to be turned on so you can test them. Ordinarily these checks are done only when running `setuid` or `setgid`. It's a good idea to turn them on explicitly for programs that run on behalf of someone else whom you might not necessarily trust, such as CGI programs or any internet servers you might write in Perl. See [perlsec](#) for details. For security reasons, this option must be seen by Perl quite early; usually this means it must appear early on the command line or in the `#!` line for systems which support that construct.

- u** This obsolete switch causes Perl to dump core after compiling your program. You can then in theory take this core dump and turn it into an executable file by using the **undump** program (not supplied). This speeds startup at the expense of some disk space (which you can minimize by stripping the executable). (Still, a "hello world" executable comes out to about 200K on my machine.) If you

want to execute a portion of your program before dumping, use the `dump()` operator instead. Note: availability of **undump** is platform specific and may not be available for a specific port of Perl.

This switch has been superseded in favor of the new Perl code generator backends to the compiler. See [B](#) and [B::Bytecode](#) for details.

- U** allows Perl to do unsafe operations. Currently the only "unsafe" operations are the unlinking of directories while running as superuser, and running setuid programs with fatal taint checks turned into warnings. Note that the **-w** switch (or the `$^W` variable) must be used along with this option to actually *generate* the taint-check warnings.
- v** prints the version and patchlevel of your perl executable.
- V** prints summary of the major perl configuration values and the current values of `@INC`.
- V:name**

Prints to STDOUT the value of the named configuration variable. For example,

```
$ perl -V:man.dir
```

will provide strong clues about what your MANPATH variable should be set to in order to access the Perl documentation.

- w** prints warnings about dubious constructs, such as variable names that are mentioned only once and scalar variables that are used before being set, redefined subroutines, references to undefined filehandles or filehandles opened read-only that you are attempting to write on, values used as a number that doesn't look like numbers, using an array as though it were a scalar, if your subroutines recurse more than 100 deep, and innumerable other things.

This switch really just enables the internal `^$W` variable. You can disable or promote into fatal errors specific warnings using `__WARN__` hooks, as described in [perlvar](#) and [warn](#). See also [perldiag](#) and [perltrap](#). A new, fine-grained warning facility is also available if you want to manipulate entire classes of warnings; see [warnings](#) or [perllexwarn](#).

- W** Enables all warnings regardless of `no warnings` or `$^W`. See [perllexwarn](#).
- X** Disables all warnings regardless of `use warnings` or `$^W`. See [perllexwarn](#).

-x directory

tells Perl that the program is embedded in a larger chunk of unrelated ASCII text, such as in a mail message. Leading garbage will be discarded until the first line that starts with `#!` and contains the string "perl". Any meaningful switches on that line will be applied. If a directory name is specified, Perl will switch to that directory before running the program. The **-x** switch controls only the disposal of leading garbage. The program must be terminated with `__END__` if there is trailing garbage to be ignored (the program can process any or all of the trailing garbage via the DATA filehandle if desired).

ENVIRONMENT

HOME	Used if <code>chdir</code> has no argument.
LOGDIR	Used if <code>chdir</code> has no argument and HOME is not set.
PATH	Used in executing subprocesses, and in finding the program if -S is used.
PERL5LIB	A colon-separated list of directories in which to look for Perl library files before looking in the standard library and the current directory. Any architecture-specific directories under the specified locations are automatically included if they exist. If PERL5LIB is not defined, PERLLIB is used.

When running taint checks (either because the program was running setuid or setgid, or the **-T** switch was used), neither variable is used. The program should instead say:

```
use lib "/my/directory";
```

PERL5OPT Command-line options (switches). Switches in this variable are taken as if they were on every Perl command line. Only the **-[DIMUdmtw]** switches are allowed. When running taint checks (because the program was running `setuid` or `setgid`, or the **-T** switch was used), this variable is ignored. If **PERL5OPT** begins with **-T**, tainting will be enabled, and any subsequent options ignored.

PERLIO A space (or colon) separated list of PerlIO layers. If perl is built to use PerlIO system for IO (the default) these layers effect perl's IO.

It is conventional to start layer names with a colon e.g. `:perlio` to emphasise their similarity to variable "attributes". But the code that parses layer specification strings (which is also used to decode the **PERLIO** environment variable) treats the colon as a separator.

The list becomes the default for *all* perl's IO. Consequently only built-in layers can appear in this list, as external layers (such as `:encoding()`) need IO in order to load them!. See "[open pragma](#)"/[open](#) for how to add external encodings as defaults.

The layers that it makes sense to include in the **PERLIO** environment variable are summarised below. For more details see [PerlIO](#).

:bytes Turns *off* the `:utf8` flag for the layer below. Unlikely to be useful in global **PERLIO** environment variable.

:crlf A layer that implements DOS/Windows like CRLF line endings. On read converts pairs of CR,LF to a single `"\n"` newline character. On write converts each `"\n"` to a CR,LF pair. Based on the `:perlio` layer.

:mmap A layer which implements "reading" of files by using `mmap()` to make (whole) file appear in the process's address space, and then using that as PerlIO's "buffer". This *may* be faster in certain circumstances for large files, and may result in less physical memory use when multiple processes are reading the same file.

Files which are not `mmap()`-able revert to behaving like the `:perlio` layer. Writes also behave like `:perlio` layer as `mmap()` for write needs extra house-keeping (to extend the file) which negates any advantage.

The `:mmap` layer will not exist if platform does not support `mmap()`.

:perlio A from scratch implementation of buffering for PerlIO. Provides fast access to the buffer for `sv_gets` which implements perl's `readline/<>` and in general attempts to minimize data copying.

`:perlio` will insert a `:unix` layer below itself to do low level IO.

:raw Applying the `<:raw` layer is equivalent to calling `binmode($fh)`. It makes the stream pass each byte as-is without any translation. In particular CRLF translation, and/or `:utf8` inited from locale are disabled.

Arranges for all accesses go straight to the lowest buffered layer provided by the configuration. That is it strips off any layers above that layer.

In Perl 5.6 and some books the `:raw` layer (previously sometimes also referred to as a "discipline") is documented as the inverse of the `:crlf` layer. That is no longer the case – other layers which would alter binary nature of the stream are also disabled. If you want UNIX line endings on a platform that normally does CRLF translation, but still want UTF-8 or encoding defaults the appropriate thing to do is to add `:perlio` to **PERLIO** environment variable.

<code>:stdio</code>	This layer provides PerlIO interface by wrapping system's ANSI C "stdio" library calls. The layer provides both buffering and IO. Note that <code>:stdio</code> layer does <i>not</i> do CRLF translation even if that is platforms normal behaviour. You will need a <code>:crlf</code> layer above it to do that.
<code>:unix</code>	Lowest level layer which provides basic PerlIO operations in terms of UNIX/POSIX numeric file descriptor calls <code>open()</code> , <code>read()</code> , <code>write()</code> , <code>lseek()</code> , <code>close()</code>
<code>:utf8</code>	Turns on a flag on the layer below to tell perl that data sent to the stream should be converted to perl internal "utf8" form and that data from the stream should be considered as so encoded. On ASCII based platforms the encoding is UTF-8 and on EBCDIC platforms UTF-EBCDIC. May be useful in PERLIO environment variable to make UTF-8 the default. (To turn off that behaviour use <code>:bytes</code> layer.)
<code>:win32</code>	On Win32 platforms this <i>experimental</i> layer uses native "handle" IO rather than unix-like numeric file descriptor layer. Known to be buggy in this release.

On all platforms the default set of layers should give acceptable results.

For UNIX platforms that will equivalent of "unix perl" or "stdio". Configure is setup to prefer "stdio" implementation if system's library provides for fast access to the buffer, otherwise it uses the "unix perl" implementation.

On Win32 the default in this release is "unix crlf". Win32's "stdio" has a number of bugs/mis-features for perl IO which are somewhat C compiler vendor/version dependent. Using our own `crlf` layer as the buffer avoids those issues and makes things more uniform. The `crlf` layer provides CRLF to/from "\n" conversion as well as buffering.

This release uses `unix` as the bottom layer on Win32 and so still uses C compiler's numeric file descriptor routines. There is an experimental native `win32` layer which is expected to be enhanced and should eventually replace the `unix` layer.

PERLIO_DEBUG

If set to the name of a file or device then certain operations of PerlIO sub-system will be logged to that file (opened as append). Typical uses are UNIX:

```
PERLIO_DEBUG=/dev/tty perl script ...
```

and Win32 approximate equivalent:

```
set PERLIO_DEBUG=CON
perl script ...
```

PERLLIB

A colon-separated list of directories in which to look for Perl library files before looking in the standard library and the current directory. If `PERL5LIB` is defined, `PERLLIB` is not used.

PERL5DB

The command used to load the debugger code. The default is:

```
BEGIN { require 'perl5db.pl' }
```

PERL5SHELL (specific to the Win32 port)

May be set to an alternative shell that perl must use internally for executing "backtick" commands or `system()`. Default is `cmd.exe /x/c` on WindowsNT and `command.com /c` on Windows95. The value is considered to be space-separated. Precede any character that needs to be protected (like a space or backslash) with a backslash.

Note that Perl doesn't use COMSPEC for this purpose because COMSPEC has a high degree of variability among users, leading to portability concerns. Besides, perl can use a shell that may not be fit for interactive use, and setting COMSPEC to such a shell may interfere with the proper functioning of other programs (which usually look in COMSPEC to find a shell fit for interactive use).

PERL_DEBUG_MSTATS

Relevant only if perl is compiled with the malloc included with the perl distribution (that is, if `perl -V:d_mymalloc` is 'define'). If set, this causes memory statistics to be dumped after execution. If set to an integer greater than one, also causes memory statistics to be dumped after compilation.

PERL_DESTRUCT_LEVEL

Relevant only if your perl executable was built with **-DDEBUGGING**, this controls the behavior of global destruction of objects and other references. See [PERL_DESTRUCT_LEVEL](#) for more information.

PERL_ENCODING

If using the encoding pragma without an explicit encoding name, the PERL_ENCODING environment variable is consulted for an encoding name.

PERL_ROOT (specific to the VMS port)

A translation concealed rooted logical name that contains perl and the logical device for the @INC path on VMS only. Other logical names that affect perl on VMS include PERLSHR, PERL_ENV_TABLES, and SYS\$TIMEZONE_DIFFERENTIAL but are optional and discussed further in [perlvms](#) and in **README.vms** in the Perl source distribution.

SYS\$LOGIN (specific to the VMS port)

Used if chdir has no argument and HOME and LOGDIR are not set.

Perl also has environment variables that control how Perl handles data specific to particular natural languages. See [perllocale](#).

Apart from these, Perl uses no other environment variables, except to make them available to the program being executed, and to child processes. However, programs running setuid would do well to execute the following lines before doing anything else, just to keep people honest:

```
$ENV{PATH} = '/bin:/usr/bin';    # or whatever you need
$ENV{SHELL} = '/bin/sh' if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

NAME

perlsec – Perl security

DESCRIPTION

Perl is designed to make it easy to program securely even when running with extra privileges, like `setuid` or `setgid` programs. Unlike most command line shells, which are based on multiple substitution passes on each line of the script, Perl uses a more conventional evaluation scheme with fewer hidden snags. Additionally, because the language has more builtin functionality, it can rely less upon external (and possibly untrustworthy) programs to accomplish its purposes.

Perl automatically enables a set of special security checks, called *taint mode*, when it detects its program running with differing real and effective user or group IDs. The `setuid` bit in Unix permissions is mode 04000, the `setgid` bit mode 02000; either or both may be set. You can also enable taint mode explicitly by using the `-T` command line flag. This flag is *strongly* suggested for server programs and any program run on behalf of someone else, such as a CGI script. Once taint mode is on, it's on for the remainder of your script.

While in this mode, Perl takes special precautions called *taint checks* to prevent both obvious and subtle traps. Some of these checks are reasonably simple, such as verifying that path directories aren't writable by others; careful programmers have always used checks like these. Other checks, however, are best supported by the language itself, and it is these checks especially that contribute to making a set-id Perl program more secure than the corresponding C program.

You may not use data derived from outside your program to affect something else outside your program—at least, not by accident. All command line arguments, environment variables, locale information (see [perllocale](#)), results of certain system calls (`readdir()`, `readlink()`, the variable of `shmread()`, the messages returned by `msgrcv()`, the password, `gcos` and `shell` fields returned by the `getpwxxx()` calls), and all file input are marked as "tainted". Tainted data may not be used directly or indirectly in any command that invokes a sub-shell, nor in any command that modifies files, directories, or processes, **with the following exceptions**:

- If you pass more than one argument to either `system` or `exec`, the arguments are checked for taintedness **but** the operation will still be attempted, emitting an optional warning. This will be fatal in a future version of perl so do not rely on it to bypass the tainting mechanism.
- Arguments to `print` and `syswrite` are **not** checked for taintedness.
- Symbolic methods

```
$obj->$method(@args);
```

and symbolic sub references

```
&{$foo}(@args);
$foo->(@args);
```

are not checked for taintedness. This requires extra carefulness unless you want external data to affect your control flow. Unless you carefully limit what these symbolic values are, people are able to call functions **outside** your Perl code, such as `POSIX::system`, in which case they are able to run arbitrary external code.

The value of an expression containing tainted data will itself be tainted, even if it is logically impossible for the tainted data to affect the value.

Because taintedness is associated with each scalar value, some elements of an array can be tainted and others not.

For example:

```
$arg = shift;           # $arg is tainted
$hid = $arg, 'bar';     # $hid is also tainted
$line = <>;             # Tainted
```

```

$line = <STDIN>;      # Also tainted
open FOO, "/home/me/bar" or die $!;
$line = <FOO>;        # Still tainted
$path = $ENV{'PATH'}; # Tainted, but see below
$data = 'abc';        # Not tainted

system "echo $arg";    # Insecure
system "/bin/echo", $arg; # Allowed but considered insecure
                        # (Perl doesn't know about /bin/echo)

system "echo $hid";    # Insecure
system "echo $data";   # Insecure until PATH set

$path = $ENV{'PATH'}; # $path now tainted

$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

$path = $ENV{'PATH'}; # $path now NOT tainted
system "echo $data";   # Is secure now!

open(FOO, "< $arg");    # OK - read-only file
open(FOO, "> $arg");    # Not OK - trying to write

open(FOO, "echo $arg|"); # Not OK
open(FOO, "-|")
    or exec 'echo', $arg; # Allowed but not really OK

$shout = `echo $arg`;   # Insecure, $shout now tainted

unlink $data, $arg;     # Insecure
umask $arg;             # Insecure

exec "echo $arg";       # Insecure
exec "echo", $arg;      # Allowed but considered insecure
exec "sh", '-c', $arg;  # Considered secure, alas!

@files = <*.c>;         # insecure (uses readdir() or similar)
@files = glob('*.c');   # insecure (uses readdir() or similar)

# In Perl releases older than 5.6.0 the <*.c> and glob('*.c') would
# have used an external program to do the filename expansion; but in
# either case the result is tainted since the list of filenames comes
# from outside of the program.

$bad = ($arg, 23);      # $bad will be tainted
$arg, `true`;           # Insecure (although it isn't really)

```

If you try to do something insecure, you will get a fatal error saying something like "Insecure dependency" or "Insecure \$ENV{PATH}". Note that you can still write an insecure **system** or **exec**, but only by explicitly doing something like the "considered secure" example above. This will not be possible in a future version of Perl.

Laundering and Detecting Tainted Data

To test whether a variable contains tainted data, and whose use would thus trigger an "Insecure dependency" message, you can use the `tainted()` function of the `Scalar::Util` module, available in your nearby CPAN mirror, and included in Perl starting from the release 5.8.0. Or you may be able to use the following `is_tainted()` function.

```

sub is_tainted {
    return ! eval { eval("#" . substr(join("", @_), 0, 0)); 1 };
}

```


This function makes use of the fact that the presence of tainted data anywhere within an expression renders the entire expression tainted. It would be inefficient for every operator to test every argument for taintedness. Instead, the slightly more efficient and conservative approach is used that if any tainted value has been accessed within the same expression, the whole expression is considered tainted.

But testing for taintedness gets you only so far. Sometimes you have just to clear your data's taintedness. The only way to bypass the tainting mechanism is by referencing subpatterns from a regular expression match. Perl presumes that if you reference a substring using `$1`, `$2`, etc., that you knew what you were doing when you wrote the pattern. That means using a bit of thought—don't just blindly untaint anything, or you defeat the entire mechanism. It's better to verify that the variable has only good characters (for certain values of "good") rather than checking whether it has any bad characters. That's because it's far too easy to miss bad characters that you never thought of.

Here's a test to make sure that the data contains nothing but "word" characters (alphanumerics, and underscores), a hyphen, an at sign, or a dot.

```
if ($data =~ /^([-@\w.]+)$/) {
    $data = $1;                # $data now untainted
} else {
    die "Bad data in $data";    # log this somewhere
}
```

This is fairly secure because `/\w+/` doesn't normally match shell metacharacters, nor are dot, dash, or at going to mean something special to the shell. Use of `/.+/` would have been insecure in theory because it lets everything through, but Perl doesn't check for that. The lesson is that when untainting, you must be exceedingly careful with your patterns. Laundering data using regular expression is the *only* mechanism for untainting dirty data, unless you use the strategy detailed below to fork a child of lesser privilege.

The example does not untaint `$data` if `use locale` is in effect, because the characters matched by `\w` are determined by the locale. Perl considers that locale definitions are untrustworthy because they contain data from outside the program. If you are writing a locale-aware program, and want to launder data with a regular expression containing `\w`, put `no locale` ahead of the expression in the same block. See [SECURITY](#) for further discussion and examples.

Switches On the "#!" Line

When you make a script executable, in order to make it usable as a command, the system will pass switches to perl from the script's `#!` line. Perl checks that any command line switches given to a `setuid` (or `setgid`) script actually match the ones set on the `#!` line. Some Unix and Unix-like environments impose a one-switch limit on the `#!` line, so you may need to use something like `-wU` instead of `-w -U` under such systems. (This issue should arise only in Unix or Unix-like environments that support `#!` and `setuid` or `setgid` scripts.)

Cleaning Up Your Path

For "Insecure `$ENV{PATH}`" messages, you need to set `$ENV{'PATH'}` to a known value, and each directory in the path must be non-writable by others than its owner and group. You may be surprised to get this message even if the pathname to your executable is fully qualified. This is *not* generated because you didn't supply a full path to the program; instead, it's generated because you never set your `PATH` environment variable, or you didn't set it to something that was safe. Because Perl can't guarantee that the executable in question isn't itself going to turn around and execute some other program that is dependent on your `PATH`, it makes sure you set the `PATH`.

The `PATH` isn't the only environment variable which can cause problems. Because some shells may use the variables `IFS`, `CDPATH`, `ENV`, and `BASH_ENV`, Perl checks that those are either empty or untainted when starting subprocesses. You may wish to add something like this to your `setid` and `taint-checking` scripts.

```
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)}; # Make %ENV safer
```

It's also possible to get into trouble with other operations that don't care whether they use tainted values. Make judicious use of the file tests in dealing with any user-supplied filenames. When possible, do opens

and such **after** properly dropping any special user (or group!) privileges. Perl doesn't prevent you from opening tainted filenames for reading, so be careful what you print out. The tainting mechanism is intended to prevent stupid mistakes, not to remove the need for thought.

Perl does not call the shell to expand wild cards when you pass **system** and **exec** explicit parameter lists instead of strings with possible shell wildcards in them. Unfortunately, the **open**, **glob**, and backtick functions provide no such alternate calling convention, so more subterfuge will be required.

Perl provides a reasonably safe way to open a file or pipe from a setuid or setgid program: just create a child process with reduced privilege who does the dirty work for you. First, fork a child using the special **open** syntax that connects the parent and child by a pipe. Now the child resets its ID set and any other per-process attributes, like environment variables, umasks, current working directories, back to the originals or known safe values. Then the child process, which no longer has any special permissions, does the **open** or other system call. Finally, the child passes the data it managed to access back to the parent. Because the file or pipe was opened in the child while running under less privilege than the parent, it's not apt to be tricked into doing something it shouldn't.

Here's a way to do backticks reasonably safely. Notice how the **exec** is not called with a string that the shell could expand. This is by far the best way to call something that might be subjected to shell escapes: just never call the shell at all.

```
use English '-no_match_vars';
die "Can't fork: $!" unless defined($pid = open(KID, "-|"));
if ($pid) {
    # parent
    while (<KID>) {
        # do something
    }
    close KID;
} else {
    my @temp      = ($EUID, $EGID);
    my $orig_uid = $UID;
    my $orig_gid = $GID;
    $EUID = $UID;
    $EGID = $GID;
    # Drop privileges
    $UID = $orig_uid;
    $GID = $orig_gid;
    # Make sure privs are really gone
    ($EUID, $EGID) = @temp;
    die "Can't drop privileges"
        unless $UID == $EUID && $GID eq $EGID;
    $ENV{PATH} = "/bin:/usr/bin"; # Minimal PATH.
    # Consider sanitizing the environment even more.
    exec 'myprog', 'arg1', 'arg2'
        or die "can't exec myprog: $!";
}
```

A similar strategy would work for wildcard expansion via **glob**, although you can use **readdir** instead.

Taint checking is most useful when although you trust yourself not to have written a program to give away the farm, you don't necessarily trust those who end up using it not to try to trick it into doing something bad. This is the kind of security checking that's useful for set-id programs and programs launched on someone else's behalf, like CGI programs.

This is quite different, however, from not even trusting the writer of the code not to try to do something evil. That's the kind of trust needed when someone hands you a program you've never seen before and says, "Here, run this." For that kind of safety, check out the **Safe** module, included standard in the Perl distribution. This module allows the programmer to set up special compartments in which all system

operations are trapped and namespace access is carefully controlled.

Security Bugs

Beyond the obvious problems that stem from giving special privileges to systems as flexible as scripts, on many versions of Unix, set-id scripts are inherently insecure right from the start. The problem is a race condition in the kernel. Between the time the kernel opens the file to see which interpreter to run and when the (now-set-id) interpreter turns around and reopens the file to interpret it, the file in question may have changed, especially if you have symbolic links on your system.

Fortunately, sometimes this kernel "feature" can be disabled. Unfortunately, there are two ways to disable it. The system can simply outlaw scripts with any set-id bit set, which doesn't help much. Alternately, it can simply ignore the set-id bits on scripts. If the latter is true, Perl can emulate the `setuid` and `setgid` mechanism when it notices the otherwise useless `setuid/gid` bits on Perl scripts. It does this via a special executable called **suidperl** that is automatically invoked for you if it's needed.

However, if the kernel set-id script feature isn't disabled, Perl will complain loudly that your set-id script is insecure. You'll need to either disable the kernel set-id script feature, or put a C wrapper around the script. A C wrapper is just a compiled program that does nothing except call your Perl program. Compiled programs are not subject to the kernel bug that plagues set-id scripts. Here's a simple wrapper, written in C:

```
#define REAL_PATH "/path/to/script"
main(ac, av)
    char **av;
{
    execv(REAL_PATH, av);
}
```

Compile this wrapper into a binary executable and then make *it* rather than your script `setuid` or `setgid`.

In recent years, vendors have begun to supply systems free of this inherent security bug. On such systems, when the kernel passes the name of the set-id script to open to the interpreter, rather than using a pathname subject to meddling, it instead passes `/dev/fd/3`. This is a special file already opened on the script, so that there can be no race condition for evil scripts to exploit. On these systems, Perl should be compiled with `-DSETUID_SCRIPTS_ARE_SECURE_NOW`. The **Configure** program that builds Perl tries to figure this out for itself, so you should never have to specify this yourself. Most modern releases of SysVr4 and BSD 4.4 use this approach to avoid the kernel race condition.

Prior to release 5.6.1 of Perl, bugs in the code of **suidperl** could introduce a security hole.

Protecting Your Programs

There are a number of ways to hide the source to your Perl programs, with varying levels of "security".

First of all, however, you *can't* take away read permission, because the source code has to be readable in order to be compiled and interpreted. (That doesn't mean that a CGI script's source is readable by people on the web, though.) So you have to leave the permissions at the socially friendly 0755 level. This lets people on your local system only see your source.

Some people mistakenly regard this as a security problem. If your program does insecure things, and relies on people not knowing how to exploit those insecurities, it is not secure. It is often possible for someone to determine the insecure things and exploit them without viewing the source. Security through obscurity, the name for hiding your bugs instead of fixing them, is little security indeed.

You can try using encryption via source filters (Filter::* from CPAN, or Filter::Util::Call and Filter::Simple since Perl 5.8). But crackers might be able to decrypt it. You can try using the byte code compiler and interpreter described below, but crackers might be able to de-compile it. You can try using the native-code compiler described below, but crackers might be able to disassemble it. These pose varying degrees of difficulty to people wanting to get at your code, but none can definitively conceal it (this is true of every language, not just Perl).

If you're concerned about people profiting from your code, then the bottom line is that nothing but a restrictive licence will give you legal security. License your software and pepper it with threatening statements like "This is unpublished proprietary software of XYZ Corp. Your access to it does not give you permission to use it blah blah blah." You should see a lawyer to be sure your licence's wording will stand up in court.

Unicode

Unicode is a new and complex technology and one may easily overlook certain security pitfalls. See [perluniintro](#) for an overview and [perlunicode](#) for details, and *[Security Implications of Unicode in perlunicode](#)* for security implications in particular.

SEE ALSO

[perlrun](#) for its description of cleaning up environment variables.

NAME

perlstyle – Perl style guide

DESCRIPTION

Each programmer will, of course, have his or her own preferences in regards to formatting, but there are some general guidelines that will make your programs easier to read, understand, and maintain.

The most important thing is to run your programs under the `-w` flag at all times. You may turn it off explicitly for particular portions of code via the `no warnings` pragma or the `$^W` variable if you must. You should also always run under `use strict` or know the reason why not. The `use sigtrap` and even `use diagnostics` pragmas may also prove useful.

Regarding aesthetics of code lay out, about the only thing Larry cares strongly about is that the closing curly bracket of a multi-line BLOCK should line up with the keyword that started the construct. Beyond that, he has other preferences that aren't so strong:

- 4-column indent.
- Opening curly on same line as keyword, if possible, otherwise line up.
- Space before the opening curly of a multi-line BLOCK.
- One-line BLOCK may be put on one line, including curlies.
- No space before the semicolon.
- Semicolon omitted in "short" one-line BLOCK.
- Space around most operators.
- Space around a "complex" subscript (inside brackets).
- Blank lines between chunks that do different things.
- Uncuddled elses.
- No space between function name and its opening parenthesis.
- Space after each comma.
- Long lines broken after an operator (except "and" and "or").
- Space after last parenthesis matching on current line.
- Line up corresponding items vertically.
- Omit redundant punctuation as long as clarity doesn't suffer.

Larry has his reasons for each of these things, but he doesn't claim that everyone else's mind works the same as his does.

Here are some other more substantive style issues to think about:

- Just because you *CAN* do something a particular way doesn't mean that you *SHOULD* do it that way. Perl is designed to give you several ways to do anything, so consider picking the most readable one. For instance

```
open(FOO,$foo) || die "Can't open $foo: $!";
```

is better than

```
die "Can't open $foo: $!" unless open(FOO,$foo);
```

because the second way hides the main point of the statement in a modifier. On the other hand

```
print "Starting analysis\n" if $verbose;
```

is better than

```
$verbose && print "Starting analysis\n";
```

because the main point isn't whether the user typed `-v` or not.

Similarly, just because an operator lets you assume default arguments doesn't mean that you have to make use of the defaults. The defaults are there for lazy systems programmers writing one-shot programs. If you want your program to be readable, consider supplying the argument.

Along the same lines, just because you *CAN* omit parentheses in many places doesn't mean that you ought to:

```
return print reverse sort num values %array;
return print(reverse(sort num (values(%array))));
```

When in doubt, parenthesize. At the very least it will let some poor schmuck bounce on the % key in vi.

Even if you aren't in doubt, consider the mental welfare of the person who has to maintain the code after you, and who will probably put parentheses in the wrong place.

- Don't go through silly contortions to exit a loop at the top or the bottom, when Perl provides the last operator so you can exit in the middle. Just "outdent" it a little to make it more visible:

```
LINE:
    for (;;) {
        statements;
        last LINE if $foo;
        next LINE if /^#/;
        statements;
    }
```

- Don't be afraid to use loop labels—they're there to enhance readability as well as to allow multilevel loop breaks. See the previous example.
- Avoid using `grep()` (or `map()`) or 'backticks' in a void context, that is, when you just throw away their return values. Those functions all have return values, so use them. Otherwise use a `foreach()` loop or the `system()` function instead.
- For portability, when using features that may not be implemented on every machine, test the construct in an eval to see if it fails. If you know what version or patchlevel a particular feature was implemented, you can test `$] ($PERL_VERSION in English)` to see if it will be there. The `Config` module will also let you interrogate values determined by the **Configure** program when Perl was installed.
- Choose mnemonic identifiers. If you can't remember what mnemonic means, you've got a problem.
- While short identifiers like `$gotit` are probably ok, use underscores to separate words. It is generally easier to read `$var_names_like_this` than `$VarNamesLikeThis`, especially for non-native speakers of English. It's also a simple rule that works consistently with `VAR_NAMES_LIKE_THIS`.

Package names are sometimes an exception to this rule. Perl informally reserves lowercase module names for "pragma" modules like `integer` and `strict`. Other modules should begin with a capital letter and use mixed case, but probably without underscores due to limitations in primitive file systems' representations of module names as files that must fit into a few sparse bytes.

- You may find it helpful to use letter case to indicate the scope or nature of a variable. For example:

```
$ALL_CAPS_HERE    constants only (beware clashes with perl vars!)
$Some_Caps_Here   package-wide global/static
$no_caps_here     function scope my() or local() variables
```

Function and method names seem to work best as all lowercase. E.g., `$obj->as_string()`.

You can use a leading underscore to indicate that a variable or function should not be used outside the package that defined it.

- If you have a really hairy regular expression, use the `/x` modifier and put in some whitespace to make it look a little less like line noise. Don't use slash as a delimiter when your regexp has slashes or backslashes.
- Use the new "and" and "or" operators to avoid having to parenthesize list operators so much, and to reduce the incidence of punctuation operators like `&&` and `||`. Call your subroutines as if they were functions or list operators to avoid excessive ampersands and parentheses.
- Use here documents instead of repeated `print()` statements.
- Line up corresponding things vertically, especially if it'd be too long to fit on one line anyway.

```
$IDX = $ST_MTIME;
$IDX = $ST_ETIME      if $opt_u;
$IDX = $ST_CTIME      if $opt_c;
$IDX = $ST_SIZE       if $opt_s;

mkdir $tmpdir, 0700 or die "can't mkdir $tmpdir: $!";
chdir($tmpdir)      or die "can't chdir $tmpdir: $!";
mkdir 'tmp', 0777 or die "can't mkdir $tmpdir/tmp: $!";
```

- Always check the return codes of system calls. Good error messages should go to `STDERR`, include which program caused the problem, what the failed system call and arguments were, and (VERY IMPORTANT) should contain the standard system error message for what went wrong. Here's a simple but sufficient example:

```
opendir(D, $dir)      or die "can't opendir $dir: $!";
```

- Line up your transliterations when it makes sense:

```
tr [abc]
   [xyz];
```

- Think about reusability. Why waste brainpower on a one-shot when you might want to do something like it again? Consider generalizing your code. Consider writing a module or object class. Consider making your code run cleanly with `use strict` and `use warnings` (or `-w`) in effect. Consider giving away your code. Consider changing your whole world view. Consider... oh, never mind.
- Be consistent.
- Be nice.

NAME

perlsub – Perl subroutines

SYNOPSIS

To declare subroutines:

```
sub NAME;                # A "forward" declaration.
sub NAME (PROTO) ;       # ditto, but with prototypes
sub NAME : ATTRS;        # with attributes
sub NAME (PROTO) : ATTRS; # with attributes and prototypes

sub NAME BLOCK           # A declaration and a definition.
sub NAME (PROTO) BLOCK   # ditto, but with prototypes
sub NAME : ATTRS BLOCK   # with attributes
sub NAME (PROTO) : ATTRS BLOCK # with prototypes and attributes
```

To define an anonymous subroutine at runtime:

```
$subref = sub BLOCK;          # no proto
$subref = sub (PROTO) BLOCK;  # with proto
$subref = sub : ATTRS BLOCK;  # with attributes
$subref = sub (PROTO) : ATTRS BLOCK; # with proto and attributes
```

To import subroutines:

```
use MODULE qw (NAME1 NAME2 NAME3);
```

To call subroutines:

```
NAME (LIST);    # & is optional with parentheses.
NAME LIST;      # Parentheses optional if predeclared/imported.
&NAME (LIST);   # Circumvent prototypes.
&NAME;          # Makes current @_ visible to called subroutine.
```

DESCRIPTION

Like many languages, Perl provides for user-defined subroutines. These may be located anywhere in the main program, loaded in from other files via the `do`, `require`, or `use` keywords, or generated on the fly using `eval` or anonymous subroutines. You can even call a function indirectly using a variable containing its name or a CODE reference.

The Perl model for function call and return values is simple: all functions are passed as parameters one single flat list of scalars, and all functions likewise return to their caller one single flat list of scalars. Any arrays or hashes in these call and return lists will collapse, losing their identities—but you may always use pass-by-reference instead to avoid this. Both call and return lists may contain as many or as few scalar elements as you'd like. (Often a function without an explicit return statement is called a subroutine, but there's really no difference from Perl's perspective.)

Any arguments passed in show up in the array `@_`. Therefore, if you called a function with two arguments, those would be stored in `$_[0]` and `$_[1]`. The array `@_` is a local array, but its elements are aliases for the actual scalar parameters. In particular, if an element `$_[0]` is updated, the corresponding argument is updated (or an error occurs if it is not updatable). If an argument is an array or hash element which did not exist when the function was called, that element is created only when (and if) it is modified or a reference to it is taken. (Some earlier versions of Perl created the element whether or not the element was assigned to.) Assigning to the whole array `@_` removes that aliasing, and does not update any arguments.

The return value of a subroutine is the value of the last expression evaluated. More explicitly, a `return` statement may be used to exit the subroutine, optionally specifying the returned value, which will be evaluated in the appropriate context (list, scalar, or void) depending on the context of the subroutine call. If you specify no return value, the subroutine returns an empty list in list context, the undefined value in scalar context, or nothing in void context. If you return one or more aggregates (arrays and hashes), these will be

flattened together into one large indistinguishable list.

Perl does not have named formal parameters. In practice all you do is assign to a `my()` list of these. Variables that aren't declared to be private are global variables. For gory details on creating private variables, see *"Private Variables via `my()`"* and *"Temporary Values via `local()`"*. To create protected environments for a set of functions in a separate package (and probably a separate file), see *Packages in `perlmod`*.

Example:

```
sub max {
    my $max = shift(@_);
    foreach $foo (@_) {
        $max = $foo if $max < $foo;
    }
    return $max;
}
$bestday = max($mon,$tue,$wed,$thu,$fri);
```

Example:

```
# get a line, combining continuation lines
# that start with whitespace

sub get_line {
    $thisline = $lookahead; # global variables!
    LINE: while (defined($lookahead = <STDIN>)) {
        if ($lookahead =~ /^[ \t]/) {
            $thisline .= $lookahead;
        }
        else {
            last LINE;
        }
    }
    return $thisline;
}

$lookahead = <STDIN>; # get first line
while (defined($line = get_line())) {
    ...
}
```

Assigning to a list of private variables to name your arguments:

```
sub maybeset {
    my($key, $value) = @_;
    $Foo{$key} = $value unless $Foo{$key};
}
```

Because the assignment copies the values, this also has the effect of turning call-by-reference into call-by-value. Otherwise a function is free to do in-place modifications of `@_` and change its caller's values.

```
upcase_in($v1, $v2); # this changes $v1 and $v2
sub upcase_in {
    for (@_) { tr/a-z/A-Z/ }
}
```

You aren't allowed to modify constants in this way, of course. If an argument were actually literal and you tried to change it, you'd take a (presumably fatal) exception. For example, this won't work:

```
upcase_in("frederick");
```

It would be much safer if the `upcase_in()` function were written to return a copy of its parameters instead of changing them in place:

```
($v3, $v4) = upcase($v1, $v2); # this doesn't change $v1 and $v2
sub upcase {
    return unless defined wantarray; # void context, do nothing
    my @parms = @_;
    for (@parms) { tr/a-z/A-Z/ }
    return wantarray ? @parms : $parms[0];
}
```

Notice how this (unprototyped) function doesn't care whether it was passed real scalars or arrays. Perl sees all arguments as one big, long, flat parameter list in `@_`. This is one area where Perl's simple argument-passing style shines. The `upcase()` function would work perfectly well without changing the `upcase()` definition even if we fed it things like this:

```
@newlist = upcase(@list1, @list2);
@newlist = upcase( split /:/, $var );
```

Do not, however, be tempted to do this:

```
(@a, @b) = upcase(@list1, @list2);
```

Like the flattened incoming parameter list, the return list is also flattened on return. So all you have managed to do here is stored everything in `@a` and made `@b` empty. See [Pass by Reference](#) for alternatives.

A subroutine may be called using an explicit `&` prefix. The `&` is optional in modern Perl, as are parentheses if the subroutine has been predeclared. The `&` is *not* optional when just naming the subroutine, such as when it's used as an argument to `defined()` or `undef()`. Nor is it optional when you want to do an indirect subroutine call with a subroutine name or reference using the `&$subref()` or `&{$subref}()` constructs, although the `< $subref-` notation solves that problem. See [perlref](#) for more about all that.

Subroutines may be called recursively. If a subroutine is called using the `&` form, the argument list is optional, and if omitted, no `@_` array is set up for the subroutine: the `@_` array at the time of the call is visible to subroutine instead. This is an efficiency mechanism that new users may wish to avoid.

```
&foo(1,2,3);      # pass three arguments
foo(1,2,3);       # the same

foo();            # pass a null list
&foo();          # the same

&foo;            # foo() get current args, like foo(@_) !!
foo;             # like foo() IFF sub foo predeclared, else "foo"
```

Not only does the `&` form make the argument list optional, it also disables any prototype checking on arguments you do provide. This is partly for historical reasons, and partly for having a convenient way to cheat if you know what you're doing. See [Prototypes](#) below.

Functions whose names are in all upper case are reserved to the Perl core, as are modules whose names are in all lower case. A function in all capitals is a loosely-held convention meaning it will be called indirectly by the run-time system itself, usually due to a triggered event. Functions that do special, pre-defined things include `BEGIN`, `CHECK`, `INIT`, `END`, `AUTOLOAD`, `CLONE` and `DESTROY`—plus all functions mentioned in [perltie](#).

Private Variables via `my()`

Synopsis:

```
my $foo;          # declare $foo lexically local
my (@wid, %get);  # declare list of variables local
```

```

my $foo = "fred"; # declare $foo lexical, and init it
my @oof = @bar; # declare @oof lexical, and init it
my $x : Foo # $x similar, with an attribute applied

```

WARNING: The use of attribute lists on my declarations is still evolving. The current semantics and interface are subject to change. See [attributes](#) and [Attribute::Handlers](#).

The my operator declares the listed variables to be lexically confined to the enclosing block, conditional (if/unless/elsif/else), loop (for/foreach/while/until/continue), subroutine, eval, or do/require/use'd file. If more than one value is listed, the list must be placed in parentheses. All listed elements must be legal lvalues. Only alphanumeric identifiers may be lexically scoped—magical built-ins like \$/ must currently be localized with local instead.

Unlike dynamic variables created by the local operator, lexical variables declared with my are totally hidden from the outside world, including any called subroutines. This is true if it's the same subroutine called from itself or elsewhere—every call gets its own copy.

This doesn't mean that a my variable declared in a statically enclosing lexical scope would be invisible. Only dynamic scopes are cut off. For example, the bumpx() function below has access to the lexical \$x variable because both the my and the sub occurred at the same scope, presumably file scope.

```

my $x = 10;
sub bumpx { $x++ }

```

An eval(), however, can see lexical variables of the scope it is being evaluated in, so long as the names aren't hidden by declarations within the eval() itself. See [perlref](#).

The parameter list to my() may be assigned to if desired, which allows you to initialize your variables. (If no initializer is given for a particular variable, it is created with the undefined value.) Commonly this is used to name input parameters to a subroutine. Examples:

```

$args = "fred";          # "global" variable
$n = cube_root(27);
print "$arg thinks the root is $n\n";
fred thinks the root is 3

sub cube_root {
    my $arg = shift; # name doesn't matter
    $arg **= 1/3;
    return $arg;
}

```

The my is simply a modifier on something you might assign to. So when you do assign to variables in its argument list, my doesn't change whether those variables are viewed as a scalar or an array. So

```

my ($foo) = <STDIN>;          # WRONG?
my @FOO = <STDIN>;

```

both supply a list context to the right-hand side, while

```

my $foo = <STDIN>;

```

supplies a scalar context. But the following declares only one variable:

```

my $foo, $bar = 1;          # WRONG

```

That has the same effect as

```

my $foo;
$bar = 1;

```

The declared variable is not introduced (is not visible) until after the current statement. Thus,

```
my $x = $x;
```

can be used to initialize a new `$x` with the value of the old `$x`, and the expression

```
my $x = 123 and $x == 123
```

is false unless the old `$x` happened to have the value 123.

Lexical scopes of control structures are not bounded precisely by the braces that delimit their controlled blocks; control expressions are part of that scope, too. Thus in the loop

```
while (my $line = <>) {
    $line = lc $line;
} continue {
    print $line;
}
```

the scope of `$line` extends from its declaration throughout the rest of the loop construct (including the `continue` clause), but not beyond it. Similarly, in the conditional

```
if ((my $answer = <STDIN>) =~ /^yes$/i) {
    user_agrees();
} elsif ($answer =~ /^no$/i) {
    user_disagrees();
} else {
    chomp $answer;
    die "'$answer' is neither 'yes' nor 'no'";
}
```

the scope of `$answer` extends from its declaration through the rest of that conditional, including any `elsif` and `else` clauses, but not beyond it.

NOTE: The behaviour of a `my` statement modified with a statement modifier conditional or loop construct (e.g. `my $x if ...`) is **undefined**. The value of the `my` variable may be `undef`, any previously assigned value, or possibly anything else. Don't rely on it. Future versions of perl might do something different from the version of perl you try it out on. Here be dragons.

The `foreach` loop defaults to scoping its index variable dynamically in the manner of `local`. However, if the index variable is prefixed with the keyword `my`, or if there is already a lexical by that name in scope, then a new lexical is created instead. Thus in the loop

```
for my $i (1, 2, 3) {
    some_function();
}
```

the scope of `$i` extends to the end of the loop, but not beyond it, rendering the value of `$i` inaccessible within `some_function()`.

Some users may wish to encourage the use of lexically scoped variables. As an aid to catching implicit uses to package variables, which are always global, if you say

```
use strict 'vars';
```

then any variable mentioned from there to the end of the enclosing block must either refer to a lexical variable, be predeclared via `our` or `use vars`, or else must be fully qualified with the package name. A compilation error results otherwise. An inner block may countermand this with `no strict 'vars'`.

A `my` has both a compile-time and a run-time effect. At compile time, the compiler takes notice of it. The principal usefulness of this is to quiet `use strict 'vars'`, but it is also essential for generation of closures as detailed in [perlref](#). Actual initialization is delayed until run time, though, so it gets executed at the appropriate time, such as each time through a loop, for example.

Variables declared with `my` are not part of any package and are therefore never fully qualified with the package name. In particular, you're not allowed to try to make a package variable (or other global) lexical:

```
my $pack::var;      # ERROR!  Illegal syntax
my $_;              # also illegal (currently)
```

In fact, a dynamic variable (also known as package or global variables) are still accessible using the fully qualified `::` notation even while a lexical of the same name is also visible:

```
package main;
local $x = 10;
my     $x = 20;
print "$x and $::x\n";
```

That will print out 20 and 10.

You may declare `my` variables at the outermost scope of a file to hide any such identifiers from the world outside that file. This is similar in spirit to C's static variables when they are used at the file level. To do this with a subroutine requires the use of a closure (an anonymous function that accesses enclosing lexicals). If you want to create a private subroutine that cannot be called from outside that block, it can declare a lexical variable containing an anonymous sub reference:

```
my $secret_version = '1.001-beta';
my $secret_sub = sub { print $secret_version };
&$secret_sub();
```

As long as the reference is never returned by any function within the module, no outside module can see the subroutine, because its name is not in any package's symbol table. Remember that it's not *REALLY* called `$some_pack::secret_version` or anything; it's just `$secret_version`, unqualified and unqualifiable.

This does not work with object methods, however; all object methods have to be in the symbol table of some package to be found. See [Function Templates in perlref](#) for something of a work-around to this.

Persistent Private Variables

Just because a lexical variable is lexically (also called statically) scoped to its enclosing block, `eval`, or `do FILE`, this doesn't mean that within a function it works like a C static. It normally works more like a C auto, but with implicit garbage collection.

Unlike local variables in C or C++, Perl's lexical variables don't necessarily get recycled just because their scope has exited. If something more permanent is still aware of the lexical, it will stick around. So long as something else references a lexical, that lexical won't be freed—which is as it should be. You wouldn't want memory being free until you were done using it, or kept around once you were done. Automatic garbage collection takes care of this for you.

This means that you can pass back or save away references to lexical variables, whereas to return a pointer to a C auto is a grave error. It also gives us a way to simulate C's function statics. Here's a mechanism for giving a function private variables with both lexical scoping and a static lifetime. If you do want to create something like C's static variables, just enclose the whole function in an extra block, and put the static variable outside the function but in the block.

```
{
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
# $secret_val now becomes unreachable by the outside
# world, but retains its value between calls to gimme_another
```

If this function is being sourced in from a separate file via `require` or `use`, then this is probably just fine. If it's all in the main program, you'll need to arrange for the `my` to be executed early, either by putting the whole block above your main program, or more likely, placing merely a `BEGIN` sub around it to make sure it gets executed before your program starts to run:

```
sub BEGIN {
    my $secret_val = 0;
    sub gimme_another {
        return ++$secret_val;
    }
}
```

See [Package Constructors and Destructors in *perlmod*](#) about the special triggered functions, `BEGIN`, `CHECK`, `INIT` and `END`.

If declared at the outermost scope (the file scope), then lexicals work somewhat like C's file statics. They are available to all functions in that same file declared below them, but are inaccessible from outside that file. This strategy is sometimes used in modules to create private variables that the whole module can see.

Temporary Values via `local()`

WARNING: In general, you should be using `my` instead of `local`, because it's faster and safer. Exceptions to this include the global punctuation variables, filehandles and formats, and direct manipulation of the Perl symbol table itself. Format variables often use `local` though, as do other variables whose current value must be visible to called subroutines.

Synopsis:

```
local $foo;                # declare $foo dynamically local
local (@wid, %get);        # declare list of variables local
local $foo = "flurp";      # declare $foo dynamic, and init it
local @oof = @bar;         # declare @oof dynamic, and init it

local *FH;                # localize $FH, @FH, %FH, &FH ...
local *merlyn = *randal;   # now $merlyn is really $randal, plus
                           #   @merlyn is really @randal, etc

local *merlyn = 'randal';  # SAME THING: promote 'randal' to *randal
local *merlyn = \"randal\"; # just alias $merlyn, not @merlyn etc
```

A `local` modifies its listed variables to be "local" to the enclosing block, `eval`, or `do FILE`—and to *any subroutine called from within that block*. A `local` just gives temporary values to global (meaning package) variables. It does *not* create a local variable. This is known as dynamic scoping. Lexical scoping is done with `my`, which works more like C's auto declarations.

If more than one variable is given to `local`, they must be placed in parentheses. All listed elements must be legal lvalues. This operator works by saving the current values of those variables in its argument list on a hidden stack and restoring them upon exiting the block, subroutine, or `eval`. This means that called subroutines can also reference the local variable, but not the global one. The argument list may be assigned to if desired, which allows you to initialize your local variables. (If no initializer is given for a particular variable, it is created with an undefined value.) Commonly this is used to name the parameters to a subroutine. Examples:

```
for $i ( 0 .. 9 ) {
    $digits{$i} = $i;
}
# assume this function uses global %digits hash
parse_num();

# now temporarily add to %digits hash
if ($base12) {
    # (NOTE: not claiming this is efficient!)
```

```

        local %digits = (%digits, 't' => 10, 'e' => 11);
        parse_num(); # parse_num gets this new %digits!
    }
    # old %digits restored here

```

Because `local` is a run-time operator, it gets executed each time through a loop. In releases of Perl previous to 5.0, this used more stack storage each time until the loop was exited. Perl now reclaims the space each time through, but it's still more efficient to declare your variables outside the loop.

A `local` is simply a modifier on an lvalue expression. When you assign to a localized variable, the `local` doesn't change whether its list is viewed as a scalar or an array. So

```

local($foo) = <STDIN>;
local @FOO = <STDIN>;

```

both supply a list context to the right-hand side, while

```

local $foo = <STDIN>;

```

supplies a scalar context.

A note about `local()` and composite types is in order. Something like `local(%foo)` works by temporarily placing a brand new hash in the symbol table. The old hash is left alone, but is hidden "behind" the new one.

This means the old variable is completely invisible via the symbol table (i.e. the hash entry in the `*foo` typeglob) for the duration of the dynamic scope within which the `local()` was seen. This has the effect of allowing one to temporarily occlude any magic on composite types. For instance, this will briefly alter a tied hash to some other implementation:

```

tie %ahash, 'APackage';
[...]
{
    local %ahash;
    tie %ahash, 'BPackage';
    [..called code will see %ahash tied to 'BPackage'..]
    {
        local %ahash;
        [..%ahash is a normal (untied) hash here..]
    }
}
[..%ahash back to its initial tied self again..]

```

WARNING The code example above does not currently work as described. This will be fixed in a future release of Perl; in the meantime, avoid code that relies on any particular behaviour of localising tied arrays or hashes (localising individual elements is still okay). See

[Localising Tied Arrays and Hashes Is Broken in perldelta](#) for more details.

As another example, a custom implementation of `%ENV` might look like this:

```

{
    local %ENV;
    tie %ENV, 'MyOwnEnv';
    [..do your own fancy %ENV manipulation here..]
}
[..normal %ENV behavior here..]

```

It's also worth taking a moment to explain what happens when you localize a member of a composite type (i.e. an array or hash element). In this case, the element is localized *by name*. This means that when the scope of the `local()` ends, the saved value will be restored to the hash element whose key was named in the `local()`, or the array element whose index was named in the `local()`. If that element was deleted

while the `local()` was in effect (e.g. by a `delete()` from a hash or a `shift()` of an array), it will spring back into existence, possibly extending an array and filling in the skipped elements with `undef`. For instance, if you say

```
%hash = ( 'This' => 'is', 'a' => 'test' );
@ary   = ( 0..5 );
{
    local($ary[5]) = 6;
    local($hash{'a'}) = 'drill';
    while (my $e = pop(@ary)) {
        print "$e . . .\n";
        last unless $e > 3;
    }
    if (@ary) {
        $hash{'only a'} = 'test';
        delete $hash{'a'};
    }
}
print join(' ', map { "$_ $hash{$_}" } sort keys %hash), ".\n";
print "The array has ", scalar(@ary), " elements: ",
      join(', ', map { defined $_ ? $_ : 'undef' } @ary), "\n";
```

Perl will print

```
6 . . .
4 . . .
3 . . .
This is a test only a test.
The array has 6 elements: 0, 1, 2, undef, undef, 5
```

The behavior of `local()` on non-existent members of composite types is subject to change in future.

Lvalue subroutines

WARNING: Lvalue subroutines are still experimental and the implementation may change in future versions of Perl.

It is possible to return a modifiable value from a subroutine. To do this, you have to declare the subroutine to return an lvalue.

```
my $val;
sub canmod : lvalue {
    # return $val; this doesn't work, don't say "return"
    $val;
}
sub nomod {
    $val;
}

canmod() = 5;    # assigns to $val
nomod()   = 5;    # ERROR
```

The scalar/list context for the subroutine and for the right-hand side of assignment is determined as if the subroutine call is replaced by a scalar. For example, consider:

```
data(2,3) = get_data(3,4);
```

Both subroutines here are called in a scalar context, while in:

```
(data(2,3)) = get_data(3,4);
```


and in:

```
(data(2),data(3)) = get_data(3,4);
```

all the subroutines are called in a list context.

Lvalue subroutines are EXPERIMENTAL

They appear to be convenient, but there are several reasons to be circumspect.

You can't use the return keyword, you must pass out the value before falling out of subroutine scope. (see comment in example above). This is usually not a problem, but it disallows an explicit return out of a deeply nested loop, which is sometimes a nice way out.

They violate encapsulation. A normal mutator can check the supplied argument before setting the attribute it is protecting, an lvalue subroutine never gets that chance. Consider;

```
my $some_array_ref = [];    # protected by mutators ??

sub set_arr {                # normal mutator
    my $val = shift;
    die("expected array, you supplied ", ref $val)
        unless ref $val eq 'ARRAY';
    $some_array_ref = $val;
}

sub set_arr_lv : lvalue {    # lvalue mutator
    $some_array_ref;
}

# set_arr_lv cannot stop this !
set_arr_lv() = { a => 1 };
```

Passing Symbol Table Entries (typeglobs)

WARNING: The mechanism described in this section was originally the only way to simulate pass-by-reference in older versions of Perl. While it still works fine in modern versions, the new reference mechanism is generally easier to work with. See below.

Sometimes you don't want to pass the value of an array to a subroutine but rather the name of it, so that the subroutine can modify the global copy of it rather than working with a local copy. In perl you can refer to all objects of a particular name by prefixing the name with a star: `*foo`. This is often known as a "typeglob", because the star on the front can be thought of as a wildcard match for all the funny prefix characters on variables and subroutines and such.

When evaluated, the typeglob produces a scalar value that represents all the objects of that name, including any filehandle, format, or subroutine. When assigned to, it causes the name mentioned to refer to whatever `*` value was assigned to it. Example:

```
sub doubleary {
    local(*someary) = @_;
    foreach $elem (@someary) {
        $elem *= 2;
    }
}

doubleary(*foo);
doubleary(*bar);
```

Scalars are already passed by reference, so you can modify scalar arguments without using this mechanism by referring explicitly to `$_[0]` etc. You can modify all the elements of an array by passing all the elements as scalars, but you have to use the `*` mechanism (or the equivalent reference mechanism) to push, pop, or change the size of an array. It will certainly be faster to pass the typeglob (or reference).

Even if you don't want to modify an array, this mechanism is useful for passing multiple arrays in a single

LIST, because normally the LIST mechanism will merge all the array values so that you can't extract out the individual arrays. For more on typeglobs, see [Typeglobs and Filehandles in perldata](#).

When to Still Use `local()`

Despite the existence of `my`, there are still three places where the `local` operator still shines. In fact, in these three places, you *must* use `local` instead of `my`.

1. You need to give a global variable a temporary value, especially `$_`.

The global variables, like `@ARGV` or the punctuation variables, must be localized with `local()`. This block reads in */etc/motd*, and splits it up into chunks separated by lines of equal signs, which are placed in `@Fields`.

```
{
    local @ARGV = ("/etc/motd");
    local $/ = undef;
    local $_ = <>;
    @Fields = split /\s*=\s*/;
}
```

It particular, it's important to localize `$_` in any routine that assigns to it. Look out for implicit assignments in `while` conditionals.

2. You need to create a local file or directory handle or a local function.

A function that needs a filehandle of its own must use `local()` on a complete typeglob. This can be used to create new symbol table entries:

```
sub ioqueue {
    local (*READER, *WRITER);    # not my!
    pipe (READER, WRITER)        or die "pipe: $!";
    return (*READER, *WRITER);
}
($head, $tail) = ioqueue();
```

See the Symbol module for a way to create anonymous symbol table entries.

Because assignment of a reference to a typeglob creates an alias, this can be used to create what is effectively a local function, or at least, a local alias.

```
{
    local *grow = \&shrink; # only until this block exists
    grow();                 # really calls shrink()
    move();                 # if move() grow()s, it shrink()s too
}
grow();                    # get the real grow() again
```

See [Function Templates in perlref](#) for more about manipulating functions by name in this way.

3. You want to temporarily change just one element of an array or hash.

You can localize just one element of an aggregate. Usually this is done on dynamics:

```
{
    local $SIG{INT} = 'IGNORE';
    funct();                                     # uninterruptible
}
# interruptibility automatically restored here
```

But it also works on lexically declared aggregates. Prior to 5.005, this operation could on occasion misbehave.

Pass by Reference

If you want to pass more than one array or hash into a function—or return them from it—and have them maintain their integrity, then you’re going to have to use an explicit pass-by-reference. Before you do that, you need to understand references as detailed in [perlref](#). This section may not make much sense to you otherwise.

Here are a few simple examples. First, let’s pass in several arrays to a function and have it pop all of them, returning a new list of all their former last elements:

```
@tailings = popmany ( \@a, \@b, \@c, \@d );

sub popmany {
    my $aref;
    my @retlist = ();
    foreach $aref ( @_ ) {
        push @retlist, pop @$aref;
    }
    return @retlist;
}
```

Here’s how you might write a function that returns a list of keys occurring in all the hashes passed to it:

```
@common = inter( \%foo, \%bar, \%joe );

sub inter {
    my ($k, $href, %seen); # locals
    foreach $href ( @_ ) {
        while ( $k = each %$href ) {
            $seen{$k}++;
        }
    }
    return grep { $seen{$_} == @_ } keys %seen;
}
```

So far, we’re using just the normal list return mechanism. What happens if you want to pass or return a hash?

Well, if you’re using only one of them, or you don’t mind them concatenating, then the normal calling convention is ok, although a little expensive.

Where people get into trouble is here:

```
(@a, @b) = func(@c, @d);
or
(%a, %b) = func(%c, %d);
```

That syntax simply won’t work. It sets just @a or %a and clears the @b or %b. Plus the function didn’t get passed into two separate arrays or hashes: it got one long list in @_, as always.

If you can arrange for everyone to deal with this through references, it’s cleaner code, although not so nice to look at. Here’s a function that takes two array references as arguments, returning the two array elements in order of how many elements they have in them:

```
($aref, $bref) = func(\@c, \@d);
print "$aref has more than @$bref\n";

sub func {
    my ($cref, $dref) = @_;
    if (@$cref > @$dref) {
        return ($cref, $dref);
    } else {
        return ($dref, $cref);
    }
}
```

```
}
```

It turns out that you can actually do this also:

```
(*a, *b) = func(\@c, \@d);
print "@a has more than @b\n";
sub func {
    local (*c, *d) = @_;
    if (@c > @d) {
        return (\@c, \@d);
    } else {
        return (\@d, \@c);
    }
}
```

Here we're using the typeglobs to do symbol table aliasing. It's a tad subtle, though, and also won't work if you're using my variables, because only globals (even in disguise as locals) are in the symbol table.

If you're passing around filehandles, you could usually just use the bare typeglob, like *STDOUT, but typeglobs references work, too. For example:

```
splutter(\*STDOUT);
sub splutter {
    my $fh = shift;
    print $fh "her um well a hmmm\n";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

If you're planning on generating new filehandles, you could do this. Notice to pass back just the bare *FH, not its reference.

```
sub openit {
    my $path = shift;
    local *FH;
    return open (FH, $path) ? *FH : undef;
}
```

Prototypes

Perl supports a very limited kind of compile-time argument checking using function prototyping. If you declare

```
sub mypush (@@)
```

then mypush() takes arguments exactly like push() does. The function declaration must be visible at compile time. The prototype affects only interpretation of new-style calls to the function, where new-style is defined as not using the & character. In other words, if you call it like a built-in function, then it behaves like a built-in function. If you call it like an old-fashioned subroutine, then it behaves like an old-fashioned subroutine. It naturally falls out from this rule that prototypes have no influence on subroutine references like \&foo or on indirect subroutine calls like &{\$subref} or < \$subref-().

Method calls are not influenced by prototypes either, because the function to be called is indeterminate at compile time, since the exact code called depends on inheritance.

Because the intent of this feature is primarily to let you define subroutines that work like built-in functions, here are prototypes for some other functions that parse almost exactly like the corresponding built-in.

Declared as	Called as
<code>sub mylink (\$\$)</code>	<code>mylink \$old, \$new</code>
<code>sub myvec (\$\$\$)</code>	<code>myvec \$var, \$offset, 1</code>
<code>sub myindex (\$\$;\$)</code>	<code>myindex &getstring, "substr"</code>
<code>sub mysyswrite (\$\$\$;\$)</code>	<code>mysyswrite \$buf, 0, length(\$buf) - \$off, \$off</code>
<code>sub myreverse (@)</code>	<code>myreverse \$a, \$b, \$c</code>
<code>sub myjoin (\$@)</code>	<code>myjoin ":", \$a, \$b, \$c</code>
<code>sub mypop (\@)</code>	<code>mypop @array</code>
<code>sub mysplICE (\@\$\$@)</code>	<code>mysplICE @array, @array, 0, @pushme</code>
<code>sub mykeys (\%)</code>	<code>mykeys %{\$hashref}</code>
<code>sub myopen (*;\$)</code>	<code>myopen HANDLE, \$name</code>
<code>sub mypipe (**)</code>	<code>mypipe READHANDLE, WRITEHANDLE</code>
<code>sub mygrep (&@)</code>	<code>mygrep { /foo/ } \$a, \$b, \$c</code>
<code>sub myrand (\$)</code>	<code>myrand 42</code>
<code>sub mytime ()</code>	<code>mytime</code>

Any backslashed prototype character represents an actual argument that absolutely must start with that character. The value passed as part of `@_` will be a reference to the actual argument given in the subroutine call, obtained by applying `\` to that argument.

You can also backslash several argument types simultaneously by using the `\[]` notation:

```
sub myref (\[$@%&*])
```

will allow calling `myref()` as

```
myref $var
myref @array
myref %hash
myref &sub
myref *glob
```

and the first argument of `myref()` will be a reference to a scalar, an array, a hash, a code, or a glob.

Unbackslashed prototype characters have special meanings. Any unbackslashed `@` or `%` eats all remaining arguments, and forces list context. An argument represented by `$` forces scalar context. An `&` requires an anonymous subroutine, which, if passed as the first argument, does not require the `sub` keyword or a subsequent comma.

A `*` allows the subroutine to accept a bareword, constant, scalar expression, typeglob, or a reference to a typeglob in that slot. The value will be available to the subroutine either as a simple scalar, or (in the latter two cases) as a reference to the typeglob. If you wish to always convert such arguments to a typeglob reference, use `Symbol::qualify_to_ref()` as follows:

```
use Symbol 'qualify_to_ref';

sub foo (*) {
    my $fh = qualify_to_ref(shift, caller);
    ...
}
```

A semicolon separates mandatory arguments from optional arguments. It is redundant before `@` or `%`, which gobble up everything else.

Note how the last three examples in the table above are treated specially by the parser. `mygrep()` is parsed as a true list operator, `myrand()` is parsed as a true unary operator with unary precedence the same as `rand()`, and `mytime()` is truly without arguments, just like `time()`. That is, if you say

```
mytime +2;
```

you'll get `mytime() + 2`, not `mytime(2)`, which is how it would be parsed without a prototype.

The interesting thing about `&` is that you can generate new syntax with it, provided it's in the initial position:

```
sub try (&@) {
    my($try,$catch) = @_;
    eval { &$try };
    if ($@) {
        local $_ = $@;
        &$catch;
    }
}
sub catch (&) { $_[0] }

try {
    die "phooey";
} catch {
    /phooey/ and print "unphooey\n";
};
```

That prints "unphooey". (Yes, there are still unresolved issues having to do with visibility of `@_`. I'm ignoring that question for the moment. (But note that if we make `@_` lexically scoped, those anonymous subroutines can act like closures... (Gee, is this sounding a little Lispish? (Never mind.))))

And here's a reimplement of the Perl `grep` operator:

```
sub mygrep (&@) {
    my $code = shift;
    my @result;
    foreach $_ (@_) {
        push(@result, $_) if &$code;
    }
    @result;
}
```

Some folks would prefer full alphanumeric prototypes. Alphanumerics have been intentionally left out of prototypes for the express purpose of someday in the future adding named, formal parameters. The current mechanism's main goal is to let module writers provide better diagnostics for module users. Larry feels the notation quite understandable to Perl programmers, and that it will not intrude greatly upon the meat of the module, nor make it harder to read. The line noise is visually encapsulated into a small pill that's easy to swallow.

If you try to use an alphanumeric sequence in a prototype you will generate an optional warning – "Illegal character in prototype...". Unfortunately earlier versions of Perl allowed the prototype to be used as long as its prefix was a valid prototype. The warning may be upgraded to a fatal error in a future version of Perl once the majority of offending code is fixed.

It's probably best to prototype new functions, not retrofit prototyping into older ones. That's because you must be especially careful about silent impositions of differing list versus scalar contexts. For example, if you decide that a function should take just one parameter, like this:

```
sub func ($) {
    my $n = shift;
    print "you gave me $n\n";
}
```

and someone has been calling it with an array or expression returning a list:

```
func(@foo);
func( split /:/ );
```

Then you've just supplied an automatic scalar in front of their argument, which can be more than a bit surprising. The old `@foo` which used to hold one thing doesn't get passed in. Instead, `func()` now gets passed in a 1; that is, the number of elements in `@foo`. And the `split` gets called in scalar context so it starts scribbling on your `@_` parameter list. Ouch!

This is all very powerful, of course, and should be used only in moderation to make the world a better place.

Constant Functions

Functions with a prototype of `()` are potential candidates for inlining. If the result after optimization and constant folding is either a constant or a lexically-scoped scalar which has no other references, then it will be used in place of function calls made without `&`. Calls made using `&` are never inlined. (See *constant.pm* for an easy way to declare most constants.)

The following functions would all be inlined:

```
sub pi ()          { 3.14159 }          # Not exact, but close.
sub PI ()          { 4 * atan2 1, 1 }    # As good as it gets,
                                         # and it's inlined, too!

sub ST_DEV ()      { 0 }
sub ST_INO ()      { 1 }

sub FLAG_FOO ()    { 1 << 8 }
sub FLAG_BAR ()    { 1 << 9 }
sub FLAG_MASK ()   { FLAG_FOO | FLAG_BAR }

sub OPT_BAZ ()     { not (0x1B58 & FLAG_MASK) }
sub BAZ_VAL () {
    if (OPT_BAZ) {
        return 23;
    }
    else {
        return 42;
    }
}

sub N () { int(BAZ_VAL) / 3 }
BEGIN {
    my $prod = 1;
    for (1..N) { $prod *= $_ }
    sub N_FACTORIAL () { $prod }
}
```

If you redefine a subroutine that was eligible for inlining, you'll get a mandatory warning. (You can use this warning to tell whether or not a particular subroutine is considered constant.) The warning is considered severe enough not to be optional because previously compiled invocations of the function will still be using the old value of the function. If you need to be able to redefine the subroutine, you need to ensure that it isn't inlined, either by dropping the `()` prototype (which changes calling semantics, so beware) or by thwarting the inlining mechanism in some other way, such as

```
sub not_inlined () {
    23 if $;
}
```

Overriding Built-in Functions

Many built-in functions may be overridden, though this should be tried only occasionally and for good reason. Typically this might be done by a package attempting to emulate missing built-in functionality on a non-Unix system.

Overriding may be done only by importing the name from a module—ordinary predeclaration isn't good enough. However, the `use subs` pragma lets you, in effect, predeclare subs via the import syntax, and

these names may then override built-in ones:

```
use subs 'chdir', 'chroot', 'chmod', 'chown';
chdir $somewhere;
sub chdir { ... }
```

To unambiguously refer to the built-in form, precede the built-in name with the special package qualifier `CORE::`. For example, saying `CORE::open()` always refers to the built-in `open()`, even if the current package has imported some other subroutine called `&open()` from elsewhere. Even though it looks like a regular function call, it isn't: you can't take a reference to it, such as the incorrect `\&CORE::open` might appear to produce.

Library modules should not in general export built-in names like `open` or `chdir` as part of their default `@EXPORT` list, because these may sneak into someone else's namespace and change the semantics unexpectedly. Instead, if the module adds that name to `@EXPORT_OK`, then it's possible for a user to import the name explicitly, but not implicitly. That is, they could say

```
use Module 'open';
```

and it would import the `open` override. But if they said

```
use Module;
```

they would get the default imports without overrides.

The foregoing mechanism for overriding built-in is restricted, quite deliberately, to the package that requests the import. There is a second method that is sometimes applicable when you wish to override a built-in everywhere, without regard to namespace boundaries. This is achieved by importing a sub into the special namespace `CORE::GLOBAL::`. Here is an example that quite brazenly replaces the `glob` operator with something that understands regular expressions.

```
package REGlob;
require Exporter;
@ISA = 'Exporter';
@EXPORT_OK = 'glob';

sub import {
    my $pkg = shift;
    return unless @_;
    my $sym = shift;
    my $where = ($sym =~ s/^\GLOBAL_// ? 'CORE::GLOBAL' : caller(0));
    $pkg->export($where, $sym, @_);
}

sub glob {
    my $pat = shift;
    my @got;
    local *D;
    if (opendir D, '.') {
        @got = grep /$pat/, readdir D;
        closedir D;
    }
    return @got;
}
1;
```

And here's how it could be (ab)used:

```
#use REGlob 'GLOBAL_glob';      # override glob() in ALL namespaces
package Foo;
use REGlob 'glob';              # override glob() in Foo:: only
```



```
print for <^[a-z_]+\\.pm\$>$show all pragmatic modules
```

The initial comment shows a contrived, even dangerous example. By overriding `glob` globally, you would be forcing the new (and subversive) behavior for the `glob` operator for *every* namespace, without the complete cognizance or cooperation of the modules that own those namespaces. Naturally, this should be done with extreme caution—if it must be done at all.

The `REGlob` example above does not implement all the support needed to cleanly override perl's `glob` operator. The built-in `glob` has different behaviors depending on whether it appears in a scalar or list context, but our `REGlob` doesn't. Indeed, many perl built-in have such context sensitive behaviors, and these must be adequately supported by a properly written override. For a fully functional example of overriding `glob`, study the implementation of `File::DosGlob` in the standard library.

When you override a built-in, your replacement should be consistent (if possible) with the built-in native syntax. You can achieve this by using a suitable prototype. To get the prototype of an overridable built-in, use the `prototype` function with an argument of `"CORE::builtin_name"` (see [prototype](#)).

Note however that some built-ins can't have their syntax expressed by a prototype (such as `system` or `chomp`). If you override them you won't be able to fully mimic their original syntax.

The built-ins `do`, `require` and `glob` can also be overridden, but due to special magic, their original syntax is preserved, and you don't have to define a prototype for their replacements. (You can't override the `do BLOCK` syntax, though).

`require` has special additional dark magic: if you invoke your `require` replacement as `require Foo::Bar`, it will actually receive the argument `"Foo/Bar.pm"` in `@_`. See [require](#).

And, as you'll have noticed from the previous example, if you override `glob`, the `<*>` `glob` operator is overridden as well.

In a similar fashion, overriding the `readline` function also overrides the equivalent I/O operator `<FILEHANDLE`.

Finally, some built-ins (e.g. `exists` or `grep`) can't be overridden.

Autoloading

If you call a subroutine that is undefined, you would ordinarily get an immediate, fatal error complaining that the subroutine doesn't exist. (Likewise for subroutines being used as methods, when the method doesn't exist in any base class of the class's package.) However, if an `AUTOLOAD` subroutine is defined in the package or packages used to locate the original subroutine, then that `AUTOLOAD` subroutine is called with the arguments that would have been passed to the original subroutine. The fully qualified name of the original subroutine magically appears in the global `$AUTOLOAD` variable of the same package as the `AUTOLOAD` routine. The name is not passed as an ordinary argument because, er, well, just because, that's why...

Many `AUTOLOAD` routines load in a definition for the requested subroutine using `eval()`, then execute that subroutine using a special form of `goto()` that erases the stack frame of the `AUTOLOAD` routine without a trace. (See the source to the standard module documented in [AutoLoader](#), for example.) But an `AUTOLOAD` routine can also just emulate the routine and never define it. For example, let's pretend that a function that wasn't defined should just invoke `system` with those arguments. All you'd do is:

```
sub AUTOLOAD {
    my $program = $AUTOLOAD;
    $program =~ s/.*:://;
    system($program, @_);
}
date();
who('am', 'i');
ls('-l');
```

In fact, if you predeclare functions you want to call that way, you don't even need parentheses:

```
use subs qw(date who ls);
date;
who "am", "i";
ls -l;
```

A more complete example of this is the standard Shell module, which can treat undefined subroutine calls as calls to external programs.

Mechanisms are available to help modules writers split their modules into autoloadable files. See the standard AutoLoader module described in [AutoLoader](#) and in [AutoSplit](#), the standard SelfLoader modules in [SelfLoader](#), and the document on adding C functions to Perl code in [perlxs](#).

Subroutine Attributes

A subroutine declaration or definition may have a list of attributes associated with it. If such an attribute list is present, it is broken up at space or colon boundaries and treated as though a `use attributes` had been seen. See [attributes](#) for details about what attributes are currently supported. Unlike the limitation with the obsolescent `use attrs`, the `sub : ATTRLIST` syntax works to associate the attributes with a pre-declaration, and not just with a subroutine definition.

The attributes must be valid as simple identifier names (without any punctuation other than the `'_'` character). They may have a parameter list appended, which is only checked for whether its parentheses `('(',')')` nest properly.

Examples of valid syntax (even though the attributes are unknown):

```
sub fnord (&%) : switch(10,foo(7,3)) : expensive ;
sub plugh () : Ugly('\(") :Bad ;
sub xyzzy : _5x5 { ... }
```

Examples of invalid syntax:

```
sub fnord : switch(10,foo() ; # ()-string not balanced
sub snoid : Ugly('(') ;      # ()-string not balanced
sub xyzzy : 5x5 ;           # "5x5" not a valid identifier
sub plugh : Y2::north ;     # "Y2::north" not a simple identifier
sub snurt : foo + bar ;     # "+" not a colon or space
```

The attribute list is passed as a list of constant strings to the code which associates them with the subroutine. In particular, the second example of valid syntax above currently looks like this in terms of how it's parsed and invoked:

```
use attributes __PACKAGE__, \&plugh, q[Ugly('\(")], 'Bad';
```

For further details on attribute lists and their manipulation, see [attributes](#) and [Attribute::Handlers](#).

SEE ALSO

See [Function Templates in perlref](#) for more about references and closures. See [perlxs](#) if you'd like to learn about calling C subroutines from Perl. See [perlembed](#) if you'd like to learn about calling Perl subroutines from C. See [perlmod](#) to learn about bundling up your functions in separate files. See [perlmodlib](#) to learn what library modules come standard on your system. See [perltoot](#) to learn how to make object method calls.

NAME

perlsyn – Perl syntax

DESCRIPTION

A Perl script consists of a sequence of declarations and statements. The sequence of statements is executed just once, unlike in **sed** and **awk** scripts, where the sequence of statements is executed for each input line. While this means that you must explicitly loop over the lines of your input file (or files), it also means you have much more control over which files and which lines you look at. (Actually, I'm lying—it is possible to do an implicit loop with either the **-n** or **-p** switch. It's just not the mandatory default like it is in **sed** and **awk**.)

Perl is, for the most part, a free-form language. (The only exception to this is format declarations, for obvious reasons.) Text from a **#** character until the end of the line is a comment, and is ignored. If you attempt to use **/* */** C-style comments, it will be interpreted either as division or pattern matching, depending on the context, and C++ **//** comments just look like a null regular expression, so don't do that.

Declarations

The only things you need to declare in Perl are report formats and subroutines—and even undefined subroutines can be handled through **AUTOLOAD**. A variable holds the undefined value (**undef**) until it has been assigned a defined value, which is anything other than **undef**. When used as a number, **undef** is treated as **0**; when used as a string, it is treated the empty string, **""**; and when used as a reference that isn't being assigned to, it is treated as an error. If you enable warnings, you'll be notified of an uninitialized value whenever you treat **undef** as a string or a number. Well, usually. Boolean contexts, such as:

```
my $a;
if ($a) {}
```

are exempt from warnings (because they care about truth rather than definedness). Operators such as **++**, **--**, **+=**, **-=**, and **.=**, that operate on undefined left values such as:

```
my $a;
$a++;
```

are also always exempt from such warnings.

A declaration can be put anywhere a statement can, but has no effect on the execution of the primary sequence of statements—declarations all take effect at compile time. Typically all the declarations are put at the beginning or the end of the script. However, if you're using lexically-scoped private variables created with **my()**, you'll have to make sure your format or subroutine definition is within the same block scope as the **my** if you expect to be able to access those private variables.

Declaring a subroutine allows a subroutine name to be used as if it were a list operator from that point forward in the program. You can declare a subroutine without defining it by saying **sub name**, thus:

```
sub myname;
$me = myname $0                or die "can't get myname";
```

Note that **myname()** functions as a list operator, not as a unary operator; so be careful to use **or** instead of **||** in this case. However, if you were to declare the subroutine as **sub myname (\$)**, then **myname** would function as a unary operator, so either **or** or **||** would work.

Subroutines declarations can also be loaded up with the **require** statement or both loaded and imported into your namespace with a **use** statement. See [perlmod](#) for details on this.

A statement sequence may contain declarations of lexically-scoped variables, but apart from declaring a variable name, the declaration acts like an ordinary statement, and is elaborated within the sequence of statements as if it were an ordinary statement. That means it actually has both compile-time and run-time effects.

Simple statements

The only kind of simple statement is an expression evaluated for its side effects. Every simple statement must be terminated with a semicolon, unless it is the final statement in a block, in which case the semicolon is optional. (A semicolon is still encouraged there if the block takes up more than one line, because you may eventually add another line.) Note that there are some operators like `eval {}` and `do {}` that look like compound statements, but aren't (they're just TERMS in an expression), and thus need an explicit termination if used as the last item in a statement.

Any simple statement may optionally be followed by a *SINGLE* modifier, just before the terminating semicolon (or block ending). The possible modifiers are:

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach EXPR
```

The `if` and `unless` modifiers have the expected semantics, presuming you're a speaker of English. The `foreach` modifier is an iterator: For each value in `EXPR`, it aliases `$_` to the value and executes the statement. The `while` and `until` modifiers have the usual "while loop" semantics (conditional evaluated first), except when applied to a `do-BLOCK` (or to the deprecated `do-SUBROUTINE` statement), in which case the block executes once before the conditional is evaluated. This is so that you can write loops like:

```
do {
    $line = <STDIN>;
    ...
} until $line eq ".\n";
```

See [do](#). Note also that the loop control statements described later will *NOT* work in this construct, because modifiers don't take loop labels. Sorry. You can always put another block inside of it (for `next`) or around it (for `last`) to do that sort of thing. For `next`, just double the braces:

```
do {{
    next if $x == $y;
    # do something here
}} until $x++ > $z;
```

For `last`, you have to be more elaborate:

```
LOOP: {
    do {
        last if $x = $y**2;
        # do something here
    } while $x++ <= $z;
}
```

Compound statements

In Perl, a sequence of statements that defines a scope is called a block. Sometimes a block is delimited by the file containing it (in the case of a required file, or the program as a whole), and sometimes a block is delimited by the extent of a string (in the case of an eval).

But generally, a block is delimited by curly brackets, also known as braces. We will call this syntactic construct a BLOCK.

The following compound statements may be used to control flow:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

```

LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK
LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK continue BLOCK
LABEL BLOCK continue BLOCK

```

Note that, unlike C and Pascal, these are defined in terms of BLOCKs, not statements. This means that the curly brackets are *required*—no dangling statements allowed. If you want to write conditionals without curly brackets there are several other ways to do it. The following all do the same thing:

```

if (!open(FOO)) { die "Can't open $FOO: $!"; }
die "Can't open $FOO: $!" unless open(FOO);
open(FOO) or die "Can't open $FOO: $!";      # FOO or bust!
open(FOO) ? 'hi mom' : die "Can't open $FOO: $!";
                        # a bit exotic, that last one

```

The `if` statement is straightforward. Because BLOCKs are always bounded by curly brackets, there is never any ambiguity about which `if` an `else` goes with. If you use `unless` in place of `if`, the sense of the test is reversed.

The `while` statement executes the block as long as the expression is true (does not evaluate to the null string "" or "0"). The LABEL is optional, and if present, consists of an identifier followed by a colon. The LABEL identifies the loop for the loop control statements `next`, `last`, and `redo`. If the LABEL is omitted, the loop control statement refers to the innermost enclosing loop. This may include dynamically looking back your call-stack at run time to find the LABEL. Such desperate behavior triggers a warning if you use the `use warnings` pragma or the `-w` flag. Unlike a `foreach` statement, a `while` statement never implicitly localises any variables.

If there is a `continue BLOCK`, it is always executed just before the conditional is about to be evaluated again, just like the third part of a `for` loop in C. Thus it can be used to increment a loop variable, even when the loop has been continued via the `next` statement (which is similar to the C `continue` statement).

Loop Control

The `next` command is like the `continue` statement in C; it starts the next iteration of the loop:

```

LINE: while (<STDIN>) {
    next LINE if /^#/;      # discard comments
    ...
}

```

The `last` command is like the `break` statement in C (as used in loops); it immediately exits the loop in question. The `continue` block, if any, is not executed:

```

LINE: while (<STDIN>) {
    last LINE if /^$/;      # exit when done with header
    ...
}

```

The `redo` command restarts the loop block without evaluating the conditional again. The `continue` block, if any, is *not* executed. This command is normally used by programs that want to lie to themselves about what was just input.

For example, when processing a file like */etc/termcap*. If your input lines might end in backslashes to indicate continuation, you want to skip ahead and get the next record.

```

while (<>) {
    chomp;
    if (s/\\$/ /) {
        $_ .= <>;
    }
}

```

```

        redo unless eof();
    }
    # now process $_
}

```

which is Perl short-hand for the more explicitly written version:

```

LINE: while (defined($line = <ARGV>)) {
    chomp($line);
    if ($line =~ s/\\$/ /) {
        $line .= <ARGV>;
        redo LINE unless eof(); # not eof(ARGV)!
    }
    # now process $line
}

```

Note that if there were a `continue` block on the above code, it would get executed only on lines discarded by the regex (since `redo` skips the `continue` block). A `continue` block is often used to reset line counters or `?pat?` one-time matches:

```

# inspired by :1,$g/fred/s//WILMA/
while (<>) {
    ?(fred)?    && s//WILMA $1 WILMA/;
    ?(barney)?  && s//BETTY $1 BETTY/;
    ?(homer)?   && s//MARGE $1 MARGE/;
} continue {
    print "$ARGV $.: $_";
    close ARGV if eof();           # reset $.
    reset      if eof();           # reset ?pat?
}

```

If the word `while` is replaced by the word `until`, the sense of the test is reversed, but the conditional is still tested before the first iteration.

The loop control statements don't work in an `if` or `unless`, since they aren't loops. You can double the braces to make them such, though.

```

if (/pattern/) {{
    last if /fred/;
    next if /barney/; # same effect as "last", but doesn't document as well
    # do something here
}}

```

This is caused by the fact that a block by itself acts as a loop that executes once, see ["Basic BLOCKs and Switch Statements"](#).

The form `while/if BLOCK BLOCK`, available in Perl 4, is no longer available. Replace any occurrence of `if BLOCK` by `if (do BLOCK)`.

For Loops

Perl's C-style `for` loop works like the corresponding `while` loop; that means that this:

```

for ($i = 1; $i < 10; $i++) {
    ...
}

```

is the same as this:

```

$i = 1;
while ($i < 10) {
    ...
}

```

```

    } continue {
        $i++;
    }

```

There is one minor difference: if variables are declared with `my` in the initialization section of the `for`, the lexical scope of those variables is exactly the `for` loop (the body of the loop and the control sections).

Besides the normal array index looping, `for` can lend itself to many other interesting applications. Here's one that avoids the problem you get into if you explicitly test for end-of-file on an interactive file descriptor causing your program to appear to hang.

```

$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "yes? " if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # do something
}

```

Foreach Loops

The `foreach` loop iterates over a normal list value and sets the variable `VAR` to be each element of the list in turn. If the variable is preceded with the keyword `my`, then it is lexically scoped, and is therefore visible only within the loop. Otherwise, the variable is implicitly local to the loop and regains its former value upon exiting the loop. If the variable was previously declared with `my`, it uses that variable instead of the global one, but it's still localized to the loop.

The `foreach` keyword is actually a synonym for the `for` keyword, so you can use `foreach` for readability or `for` for brevity. (Or because the Bourne shell is more familiar to you than *csh*, so writing `for` comes more naturally.) If `VAR` is omitted, `$_` is set to each value.

If any element of `LIST` is an lvalue, you can modify it by modifying `VAR` inside the loop. Conversely, if any element of `LIST` is NOT an lvalue, any attempt to modify that element will fail. In other words, the `foreach` loop index variable is an implicit alias for each item in the list that you're looping over.

If any part of `LIST` is an array, `foreach` will get very confused if you add or remove elements within the loop body, for example with `splice`. So don't do that.

`foreach` probably won't do what you expect if `VAR` is a tied or other special variable. Don't do that either.

Examples:

```

for (@ary) { s/foo/bar/ }
for my $elem (@elements) {
    $elem *= 2;
}
for $count (10,9,8,7,6,5,4,3,2,1,'BOOM') {
    print $count, "\n"; sleep(1);
}
for (1..15) { print "Merry Christmas\n"; }
foreach $item (split(/:[\\n:]*/, $ENV{TERMCAP})) {
    print "Item: $item\n";
}

```

Here's how a C programmer might code up a particular algorithm in Perl:

```

for (my $i = 0; $i < @ary1; $i++) {
    for (my $j = 0; $j < @ary2; $j++) {
        if ($ary1[$i] > $ary2[$j]) {
            last; # can't go to outer :- (

```

```

    }
    $ary1[$i] += $ary2[$j];
}
# this is where that last takes me
}

```

Whereas here's how a Perl programmer more comfortable with the idiom might do it:

```

OUTER: for my $wid (@ary1) {
  INNER:   for my $jet (@ary2) {
            next OUTER if $wid > $jet;
            $wid += $jet;
          }
        }

```

See how much easier this is? It's cleaner, safer, and faster. It's cleaner because it's less noisy. It's safer because if code gets added between the inner and outer loops later on, the new code won't be accidentally executed. The next explicitly iterates the other loop rather than merely terminating the inner one. And it's faster because Perl executes a `foreach` statement more rapidly than it would the equivalent `for` loop.

Basic BLOCKs and Switch Statements

A BLOCK by itself (labeled or not) is semantically equivalent to a loop that executes once. Thus you can use any of the loop control statements in it to leave or restart the block. (Note that this is *NOT* true in `eval{}`, `sub{}`, or contrary to popular belief `do{}` blocks, which do *NOT* count as loops.) The `continue` block is optional.

The BLOCK construct is particularly nice for doing case structures.

```

SWITCH: {
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}

```

There is no official `switch` statement in Perl, because there are already several ways to write the equivalent.

However, starting from Perl 5.8 to get `switch` and `case` one can use the Switch extension and say:

```
use Switch;
```

after which one has `switch` and `case`. It is not as fast as it could be because it's not really part of the language (it's done using source filters) but it is available, and it's very flexible.

In addition to the above BLOCK construct, you could write

```

SWITCH: {
    $abc = 1, last SWITCH if /^abc/;
    $def = 1, last SWITCH if /^def/;
    $xyz = 1, last SWITCH if /^xyz/;
    $nothing = 1;
}

```

(That's actually not as strange as it looks once you realize that you can use loop control "operators" within an expression, That's just the normal C comma operator.)

or

```

SWITCH: {
    /^abc/ && do { $abc = 1; last SWITCH; };
    /^def/ && do { $def = 1; last SWITCH; };
}

```



```

    /^xyz/ && do { $xyz = 1; last SWITCH; };
    $nothing = 1;
}

```

or formatted so it stands out more as a "proper" switch statement:

```

SWITCH: {
    /^abc/      && do {
                        $abc = 1;
                        last SWITCH;
                    };

    /^def/      && do {
                        $def = 1;
                        last SWITCH;
                    };

    /^xyz/      && do {
                        $xyz = 1;
                        last SWITCH;
                    };

    $nothing = 1;
}

```

or

```

SWITCH: {
    /^abc/ and $abc = 1, last SWITCH;
    /^def/ and $def = 1, last SWITCH;
    /^xyz/ and $xyz = 1, last SWITCH;
    $nothing = 1;
}

```

or even, horrors,

```

if (/^abc/)
    { $abc = 1 }
elsif (/^def/)
    { $def = 1 }
elsif (/^xyz/)
    { $xyz = 1 }
else
    { $nothing = 1 }

```

A common idiom for a switch statement is to use `foreach`'s aliasing to make a temporary assignment to `$_` for convenient matching:

```

SWITCH: for ($where) {
    /In Card Names/      && do { push @flags, '-e'; last; };
    /Anywhere/           && do { push @flags, '-h'; last; };
    /In Rulings/         && do {                               last; };
    die "unknown value for form variable where: '$where'";
}

```

Another interesting approach to a switch statement is arrange for a `do` block to return the proper value:

```

$amode = do {
    if      ($flag & O_RDONLY) { "r" }          # XXX: isn't this 0?
    elsif   ($flag & O_WRONLY) { ($flag & O_APPEND) ? "a" : "w" }
    elsif   ($flag & O_RDWR)   {

```

```

        if ($flag & O_CREAT) { "w+" }
        else                { ($flag & O_APPEND) ? "a+" : "r+" }
    }
};

```

Or

```

print do {
    ($flags & O_WRONLY) ? "write-only"      :
    ($flags & O_RDWR)   ? "read-write"     :
                        "read-only";
};

```

Or if you are certain that all the && clauses are true, you can use something like this, which "switches" on the value of the HTTP_USER_AGENT environment variable.

```

#!/usr/bin/perl
# pick out jargon file page based on browser
$dir = 'http://www.wins.uva.nl/~mes/jargon';
for ($ENV{HTTP_USER_AGENT}) {
    $page =    /Mac/           && 'm/Macintrash.html'
              || /Win(dows)?NT/ && 'e/evilandrude.html'
              || /Win|MSIE|WebTV/ && 'm/MicroslothWindows.html'
              || /Linux/        && 'l/Linux.html'
              || /HP-UX/        && 'h/HP-SUX.html'
              || /SunOS/        && 's/ScumOS.html'
              ||                'a/AppendixB.html';
}
print "Location: $dir/$page\015\012\015\012";

```

That kind of switch statement only works when you know the && clauses will be true. If you don't, the previous ? : example should be used.

You might also consider writing a hash of subroutine references instead of synthesizing a switch statement.

Goto

Although not for the faint of heart, Perl does support a goto statement. There are three forms: goto-LABEL, goto-EXPR, and goto-&NAME. A loop's LABEL is not actually a valid target for a goto; it's just the name of the loop.

The goto-LABEL form finds the statement labeled with LABEL and resumes execution there. It may not be used to go into any construct that requires initialization, such as a subroutine or a foreach loop. It also can't be used to go into a construct that is optimized away. It can be used to go almost anywhere else within the dynamic scope, including out of subroutines, but it's usually better to use some other construct such as last or die. The author of Perl has never felt the need to use this form of goto (in Perl, that is—C is another matter).

The goto-EXPR form expects a label name, whose scope will be resolved dynamically. This allows for computed gotos per FORTRAN, but isn't necessarily recommended if you're optimizing for maintainability:

```
goto(("FOO", "BAR", "GLARCH")[$i]);
```

The goto-&NAME form is highly magical, and substitutes a call to the named subroutine for the currently running subroutine. This is used by AUTOLOAD() subroutines that wish to load another subroutine and then pretend that the other subroutine had been called in the first place (except that any modifications to @_ in the current subroutine are propagated to the other subroutine.) After the goto, not even caller() will be able to tell that this routine was called first.

In almost all cases like this, it's usually a far, far better idea to use the structured control flow mechanisms of `next`, `last`, or `redo` instead of resorting to a `goto`. For certain applications, the `catch` and `throw` pair of `eval{}` and `die()` for exception processing can also be a prudent approach.

PODs: Embedded Documentation

Perl has a mechanism for intermixing documentation with source code. While it's expecting the beginning of a new statement, if the compiler encounters a line that begins with an equal sign and a word, like this

```
=head1 Here There Be Pods!
```

Then that text and all remaining text up through and including a line beginning with `=cut` will be ignored. The format of the intervening text is described in [perlpod](#).

This allows you to intermix your source code and your documentation text freely, as in

```
=item snazzle($)  
  
The snazzle() function will behave in the most spectacular  
form that you can possibly imagine, not even excepting  
cybernetic pyrotechnics.  
  
=cut back to the compiler, nuff of this pod stuff!  
  
sub snazzle($) {  
    my $thingie = shift;  
    .....  
}
```

Note that pod translators should look at only paragraphs beginning with a pod directive (it makes parsing easier), whereas the compiler actually knows to look for pod escapes even in the middle of a paragraph. This means that the following secret stuff will be ignored by both the compiler and the translators.

```
$a=3;  
=secret stuff  
    warn "Neither POD nor CODE!?"  
=cut back  
print "got $a\n";
```

You probably shouldn't rely upon the `warn()` being podded out forever. Not all pod translators are well-behaved in this regard, and perhaps the compiler will become pickier.

One may also use pod directives to quickly comment out a section of code.

Plain Old Comments (Not!)

Much like the C preprocessor, Perl can process line directives. Using this, one can control Perl's idea of filenames and line numbers in error or warning messages (especially for strings that are processed with `eval()`). The syntax for this mechanism is the same as for most C preprocessors: it matches the regular expression `/^#\s*line\s+(\d+)\s*(?:\s"([^\"]+)")?\s*$/` with `$1` being the line number for the next line, and `$2` being the optional filename (specified within quotes).

There is a fairly obvious gotcha included with the line directive: Debuggers and profilers will only show the last source line to appear at a particular line number in a given file. Care should be taken not to cause line number collisions in code you'd like to debug later.

Here are some examples that you should be able to type into your command shell:

```
% perl  
# line 200 "bzzzt"  
# the '#' on the previous line must be the first char on line  
die 'foo';  
__END__  
foo at bzzzt line 201.
```

```
% perl
# line 200 "bzzzt"
eval qq[\n#line 2001 ""\ndie 'foo']; print $@;
__END__
foo at - line 2001.

% perl
eval qq[\n#line 200 "foo bar"\ndie 'foo']; print $@;
__END__
foo at foo bar line 200.

% perl
# line 345 "goop"
eval "\n#line " . __LINE__ . ' "' . __FILE__ . "\n\ndie 'foo'";
print $@;
__END__
foo at goop line 345.
```

NAME

perlthrtut – tutorial on threads in Perl

DESCRIPTION

NOTE: this tutorial describes the new Perl threading flavour introduced in Perl 5.6.0 called interpreter threads, or **ithreads** for short. In this model each thread runs in its own Perl interpreter, and any data sharing between threads must be explicit.

There is another older Perl threading flavour called the 5.005 model, unsurprisingly for 5.005 versions of Perl. The old model is known to have problems, deprecated, and will probably be removed around release 5.10. You are strongly encouraged to migrate any existing 5.005 threads code to the new model as soon as possible.

You can see which (or neither) threading flavour you have by running `perl -V` and looking at the Platform section. If you have `useithreads=define` you have **ithreads**, if you have `use5005threads=define` you have 5.005 threads. If you have neither, you don't have any thread support built in. If you have both, you are in trouble.

The user-level interface to the 5.005 threads was via the *Threads* class, while **ithreads** uses the *threads* class. Note the change in case.

Status

The **ithreads** code has been available since Perl 5.6.0, and is considered stable. The user-level interface to **ithreads** (the *threads* classes) appeared in the 5.8.0 release, and as of this time is considered stable although it should be treated with caution as with all new features.

What Is A Thread Anyway?

A thread is a flow of control through a program with a single execution point.

Sounds an awful lot like a process, doesn't it? Well, it should. Threads are one of the pieces of a process. Every process has at least one thread and, up until now, every process running Perl had only one thread. With 5.8, though, you can create extra threads. We're going to show you how, when, and why.

Threaded Program Models

There are three basic ways that you can structure a threaded program. Which model you choose depends on what you need your program to do. For many non-trivial threaded programs you'll need to choose different models for different pieces of your program.

Boss/Worker

The boss/worker model usually has one 'boss' thread and one or more 'worker' threads. The boss thread gathers or generates tasks that need to be done, then parcels those tasks out to the appropriate worker thread.

This model is common in GUI and server programs, where a main thread waits for some event and then passes that event to the appropriate worker threads for processing. Once the event has been passed on, the boss thread goes back to waiting for another event.

The boss thread does relatively little work. While tasks aren't necessarily performed faster than with any other method, it tends to have the best user-response times.

Work Crew

In the work crew model, several threads are created that do essentially the same thing to different pieces of data. It closely mirrors classical parallel processing and vector processors, where a large array of processors do the exact same thing to many pieces of data.

This model is particularly useful if the system running the program will distribute multiple threads across different processors. It can also be useful in ray tracing or rendering engines, where the individual threads can pass on interim results to give the user visual feedback.

Pipeline

The pipeline model divides up a task into a series of steps, and passes the results of one step on to the thread processing the next. Each thread does one thing to each piece of data and passes the results to the next thread in line.

This model makes the most sense if you have multiple processors so two or more threads will be executing in parallel, though it can often make sense in other contexts as well. It tends to keep the individual tasks small and simple, as well as allowing some parts of the pipeline to block (on I/O or system calls, for example) while other parts keep going. If you're running different parts of the pipeline on different processors you may also take advantage of the caches on each processor.

This model is also handy for a form of recursive programming where, rather than having a subroutine call itself, it instead creates another thread. Prime and Fibonacci generators both map well to this form of the pipeline model. (A version of a prime number generator is presented later on.)

Native threads

There are several different ways to implement threads on a system. How threads are implemented depends both on the vendor and, in some cases, the version of the operating system. Often the first implementation will be relatively simple, but later versions of the OS will be more sophisticated.

While the information in this section is useful, it's not necessary, so you can skip it if you don't feel up to it.

There are three basic categories of threads: user-mode threads, kernel threads, and multiprocessor kernel threads.

User-mode threads are threads that live entirely within a program and its libraries. In this model, the OS knows nothing about threads. As far as it's concerned, your process is just a process.

This is the easiest way to implement threads, and the way most OSes start. The big disadvantage is that, since the OS knows nothing about threads, if one thread blocks they all do. Typical blocking activities include most system calls, most I/O, and things like `sleep()`.

Kernel threads are the next step in thread evolution. The OS knows about kernel threads, and makes allowances for them. The main difference between a kernel thread and a user-mode thread is blocking. With kernel threads, things that block a single thread don't block other threads. This is not the case with user-mode threads, where the kernel blocks at the process level and not the thread level.

This is a big step forward, and can give a threaded program quite a performance boost over non-threaded programs. Threads that block performing I/O, for example, won't block threads that are doing other things. Each process still has only one thread running at once, though, regardless of how many CPUs a system might have.

Since kernel threading can interrupt a thread at any time, they will uncover some of the implicit locking assumptions you may make in your program. For example, something as simple as `$a = $a + 2` can behave unpredictably with kernel threads if `$a` is visible to other threads, as another thread may have changed `$a` between the time it was fetched on the right hand side and the time the new value is stored.

Multiprocessor kernel threads are the final step in thread support. With multiprocessor kernel threads on a machine with multiple CPUs, the OS may schedule two or more threads to run simultaneously on different CPUs.

This can give a serious performance boost to your threaded program, since more than one thread will be executing at the same time. As a tradeoff, though, any of those nagging synchronization issues that might not have shown with basic kernel threads will appear with a vengeance.

In addition to the different levels of OS involvement in threads, different OSes (and different thread implementations for a particular OS) allocate CPU cycles to threads in different ways.

Cooperative multitasking systems have running threads give up control if one of two things happen. If a thread calls a yield function, it gives up control. It also gives up control if the thread does something that would cause it to block, such as perform I/O. In a cooperative multitasking implementation, one thread can

starve all the others for CPU time if it so chooses.

Preemptive multitasking systems interrupt threads at regular intervals while the system decides which thread should run next. In a preemptive multitasking system, one thread usually won't monopolize the CPU.

On some systems, there can be cooperative and preemptive threads running simultaneously. (Threads running with realtime priorities often behave cooperatively, for example, while threads running at normal priorities behave preemptively.)

What kind of threads are Perl threads?

If you have experience with other thread implementations, you might find that things aren't quite what you expect. It's very important to remember when dealing with Perl threads that Perl Threads Are Not X Threads, for all values of X. They aren't POSIX threads, or DecThreads, or Java's Green threads, or Win32 threads. There are similarities, and the broad concepts are the same, but if you start looking for implementation details you're going to be either disappointed or confused. Possibly both.

This is not to say that Perl threads are completely different from everything that's ever come before—they're not. Perl's threading model owes a lot to other thread models, especially POSIX. Just as Perl is not C, though, Perl threads are not POSIX threads. So if you find yourself looking for mutexes, or thread priorities, it's time to step back a bit and think about what you want to do and how Perl can do it.

However it is important to remember that Perl threads cannot magically do things unless your operating systems threads allows it. So if your system blocks the entire process on `sleep()`, Perl usually will as well.

Perl Threads Are Different.

Thread-Safe Modules

The addition of threads has changed Perl's internals substantially. There are implications for people who write modules with XS code or external libraries. However, since perl data is not shared among threads by default, Perl modules stand a high chance of being thread-safe or can be made thread-safe easily. Modules that are not tagged as thread-safe should be tested or code reviewed before being used in production code.

Not all modules that you might use are thread-safe, and you should always assume a module is unsafe unless the documentation says otherwise. This includes modules that are distributed as part of the core. Threads are a new feature, and even some of the standard modules aren't thread-safe.

Even if a module is thread-safe, it doesn't mean that the module is optimized to work well with threads. A module could possibly be rewritten to utilize the new features in threaded Perl to increase performance in a threaded environment.

If you're using a module that's not thread-safe for some reason, you can protect yourself by using it from one, and only one thread at all. If you need multiple threads to access such a module, you can use semaphores and lots of programming discipline to control access to it. Semaphores are covered in .

See also .

Thread Basics

The core [threads](#) module provides the basic functions you need to write threaded programs. In the following sections we'll cover the basics, showing you what you need to do to create a threaded program. After that, we'll go over some of the features of the [threads](#) module that make threaded programming easier.

Basic Thread Support

Thread support is a Perl compile-time option – it's something that's turned on or off when Perl is built at your site, rather than when your programs are compiled. If your Perl wasn't compiled with thread support enabled, then any attempt to use threads will fail.

Your programs can use the Config module to check whether threads are enabled. If your program can't run without them, you can say something like:

```
$Config{useithreads} or die "Recompile Perl with threads to run this program.";
```

A possibly-threaded program using a possibly-threaded module might have code like this:

```
use Config;
use MyMod;

BEGIN {
    if ($Config{useithreads}) {
        # We have threads
        require MyMod_threaded;
        import MyMod_threaded;
    } else {
        require MyMod_unthreaded;
        import MyMod_unthreaded;
    }
}
```

Since code that runs both with and without threads is usually pretty messy, it's best to isolate the thread-specific code in its own module. In our example above, that's what `MyMod_threaded` is, and it's only imported if we're running on a threaded Perl.

A Note about the Examples

Although thread support is considered to be stable, there are still a number of quirks that may startle you when you try out any of the examples below. In a real situation, care should be taken that all threads are finished executing before the program exits. That care has **not** been taken in these examples in the interest of simplicity. Running these examples "as is" will produce error messages, usually caused by the fact that there are still threads running when the program exits. You should not be alarmed by this. Future versions of Perl may fix this problem.

Creating Threads

The *threads* package provides the tools you need to create new threads. Like any other module, you need to tell Perl that you want to use it; `use threads` imports all the pieces you need to create basic threads.

The simplest, most straightforward way to create a thread is with `new()`:

```
use threads;

$thr = threads->new(\&sub1);

sub sub1 {
    print "In the thread\n";
}
```

The `new()` method takes a reference to a subroutine and creates a new thread, which starts executing in the referenced subroutine. Control then passes both to the subroutine and the caller.

If you need to, your program can pass parameters to the subroutine as part of the thread startup. Just include the list of parameters as part of the `threads::new` call, like this:

```
use threads;

$Param3 = "foo";
$thr = threads->new(\&sub1, "Param 1", "Param 2", $Param3);
$thr = threads->new(\&sub1, @ParamList);
$thr = threads->new(\&sub1, qw(Param1 Param2 Param3));

sub sub1 {
    my @InboundParameters = @_;
    print "In the thread\n";
    print "got parameters >", join("<>", @InboundParameters), "<\n";
}
```


The last example illustrates another feature of threads. You can spawn off several threads using the same subroutine. Each thread executes the same subroutine, but in a separate thread with a separate environment and potentially separate arguments.

`create()` is a synonym for `new()`.

Giving up control

There are times when you may find it useful to have a thread explicitly give up the CPU to another thread. Your threading package might not support preemptive multitasking for threads, for example, or you may be doing something processor-intensive and want to make sure that the user-interface thread gets called frequently. Regardless, there are times that you might want a thread to give up the processor.

Perl's threading package provides the `yield()` function that does this. `yield()` is pretty straightforward, and works like this:

```
use threads;

sub loop {
    my $thread = shift;
    my $foo = 50;
    while($foo-->0) { print "in thread $thread\n" }
    threads->yield;
    $foo = 50;
    while($foo-->0) { print "in thread $thread\n" }
}

my $thread1 = threads->new(\&loop, 'first');
my $thread2 = threads->new(\&loop, 'second');
my $thread3 = threads->new(\&loop, 'third');
```

It is important to remember that `yield()` is only a hint to give up the CPU, it depends on your hardware, OS and threading libraries what actually happens. Therefore it is important to note that one should not build the scheduling of the threads around `yield()` calls. It might work on your platform but it won't work on another platform.

Waiting For A Thread To Exit

Since threads are also subroutines, they can return values. To wait for a thread to exit and extract any values it might return, you can use the `join()` method:

```
use threads;

$thr = threads->new(\&sub1);

@ReturnData = $thr->join;
print "Thread returned @ReturnData";

sub sub1 { return "Fifty-six", "foo", 2; }
```

In the example above, the `join()` method returns as soon as the thread ends. In addition to waiting for a thread to finish and gathering up any values that the thread might have returned, `join()` also performs any OS cleanup necessary for the thread. That cleanup might be important, especially for long-running programs that spawn lots of threads. If you don't want the return values and don't want to wait for the thread to finish, you should call the `detach()` method instead, as described next.

Ignoring A Thread

`join()` does three things: it waits for a thread to exit, cleans up after it, and returns any data the thread may have produced. But what if you're not interested in the thread's return values, and you don't really care when the thread finishes? All you want is for the thread to get cleaned up after when it's done.

In this case, you use the `detach()` method. Once a thread is detached, it'll run until it's finished, then Perl will clean up after it automatically.

```

use threads;

$thr = threads->new(\&sub1); # Spawn the thread

$thr->detach; # Now we officially don't care any more

sub sub1 {
    $a = 0;
    while (1) {
        $a++;
        print "\$a is $a\n";
        sleep 1;
    }
}

```

Once a thread is detached, it may not be joined, and any return data that it might have produced (if it was done and waiting for a join) is lost.

Threads And Data

Now that we've covered the basics of threads, it's time for our next topic: data. Threading introduces a couple of complications to data access that non-threaded programs never need to worry about.

Shared And Unshared Data

The biggest difference between Perl ithreads and the old 5.005 style threading, or for that matter, to most other threading systems out there, is that by default, no data is shared. When a new perl thread is created, all the data associated with the current thread is copied to the new thread, and is subsequently private to that new thread! This is similar in feel to what happens when a UNIX process forks, except that in this case, the data is just copied to a different part of memory within the same process rather than a real fork taking place.

To make use of threading however, one usually wants the threads to share at least some data between themselves. This is done with the [threads::shared](#) module and the `: shared` attribute:

```

use threads;
use threads::shared;

my $foo : shared = 1;
my $bar = 1;
threads->new(sub { $foo++; $bar++ }->join;

print "$foo\n"; #prints 2 since $foo is shared
print "$bar\n"; #prints 1 since $bar is not shared

```

In the case of a shared array, all the array's elements are shared, and for a shared hash, all the keys and values are shared. This places restrictions on what may be assigned to shared array and hash elements: only simple values or references to shared variables are allowed – this is so that a private variable can't accidentally become shared. A bad assignment will cause the thread to die. For example:

```

use threads;
use threads::shared;

my $var = 1;
my $svar : shared = 2;
my %hash : shared;

... create some threads ...

$hash{a} = 1;          # all threads see exists($hash{a}) and $hash{a} == 1
$hash{a} = $var        # okay - copy-by-value: same effect as previous
$hash{a} = $svar       # okay - copy-by-value: same effect as previous
$hash{a} = \$svar      # okay - a reference to a shared variable
$hash{a} = \$var       # This will die
delete $hash{a}        # okay - all threads will see !exists($hash{a})

```

Note that a shared variable guarantees that if two or more threads try to modify it at the same time, the internal state of the variable will not become corrupted. However, there are no guarantees beyond this, as explained in the next section.

Thread Pitfalls: Races

While threads bring a new set of useful tools, they also bring a number of pitfalls. One pitfall is the race condition:

```
use threads;
use threads::shared;

my $a : shared = 1;
$thr1 = threads->new(\&sub1);
$thr2 = threads->new(\&sub2);

$thr1->join;
$thr2->join;
print "$a\n";

sub sub1 { my $foo = $a; $a = $foo + 1; }
sub sub2 { my $bar = $a; $a = $bar + 1; }
```

What do you think `$a` will be? The answer, unfortunately, is "it depends." Both `sub1()` and `sub2()` access the global variable `$a`, once to read and once to write. Depending on factors ranging from your thread implementation's scheduling algorithm to the phase of the moon, `$a` can be 2 or 3.

Race conditions are caused by unsynchronized access to shared data. Without explicit synchronization, there's no way to be sure that nothing has happened to the shared data between the time you access it and the time you update it. Even this simple code fragment has the possibility of error:

```
use threads;
my $a : shared = 2;
my $b : shared;
my $c : shared;
my $thr1 = threads->create(sub { $b = $a; $a = $b + 1; });
my $thr2 = threads->create(sub { $c = $a; $a = $c + 1; });
$thr1->join;
$thr2->join;
```

Two threads both access `$a`. Each thread can potentially be interrupted at any point, or be executed in any order. At the end, `$a` could be 3 or 4, and both `$b` and `$c` could be 2 or 3.

Even `$a += 5` or `$a++` are not guaranteed to be atomic.

Whenever your program accesses data or resources that can be accessed by other threads, you must take steps to coordinate access or risk data inconsistency and race conditions. Note that Perl will protect its internals from your race conditions, but it won't protect you from you.

Synchronization and control

Perl provides a number of mechanisms to coordinate the interactions between themselves and their data, to avoid race conditions and the like. Some of these are designed to resemble the common techniques used in thread libraries such as `pthread`; others are Perl-specific. Often, the standard techniques are clumsy and difficult to get right (such as condition waits). Where possible, it is usually easier to use Perlish techniques such as queues, which remove some of the hard work involved.

Controlling access: `lock()`

The `lock()` function takes a shared variable and puts a lock on it. No other thread may lock the variable until the the variable is unlocked by the thread holding the lock. Unlocking happens automatically when the locking thread exits the outermost block that contains `lock()` function. Using `lock()` is straightforward: this example has several threads doing some calculations in parallel, and occasionally updating a running

total:

```

use threads;
use threads::shared;

my $total : shared = 0;

sub calc {
    for (;;) {
        my $result;
        # (... do some calculations and set $result ...)
        {
            lock($total); # block until we obtain the lock
            $total += $result;
        } # lock implicitly released at end of scope
        last if $result == 0;
    }
}

my $thr1 = threads->new(\&calc);
my $thr2 = threads->new(\&calc);
my $thr3 = threads->new(\&calc);
$thr1->join;
$thr2->join;
$thr3->join;
print "total=$total\n";

```

`lock()` blocks the thread until the variable being locked is available. When `lock()` returns, your thread can be sure that no other thread can lock that variable until the outermost block containing the lock exits.

It's important to note that locks don't prevent access to the variable in question, only lock attempts. This is in keeping with Perl's longstanding tradition of courteous programming, and the advisory file locking that `flock()` gives you.

You may lock arrays and hashes as well as scalars. Locking an array, though, will not block subsequent locks on array elements, just lock attempts on the array itself.

Locks are recursive, which means it's okay for a thread to lock a variable more than once. The lock will last until the outermost `lock()` on the variable goes out of scope. For example:

```

my $x : shared;
doit();

sub doit {
    {
        {
            lock($x); # wait for lock
            lock($x); # NOOP - we already have the lock
            {
                lock($x); # NOOP
                {
                    lock($x); # NOOP
                    lockit_some_more();
                }
            }
        } # *** implicit unlock here ***
    }
}

```

```
sub lockit_some_more {
    lock($x); # NOOP
} # nothing happens here
```

Note that there is no `unlock()` function – the only way to unlock a variable is to allow it to go out of scope.

A lock can either be used to guard the data contained within the variable being locked, or it can be used to guard something else, like a section of code. In this latter case, the variable in question does not hold any useful data, and exists only for the purpose of being locked. In this respect, the variable behaves like the mutexes and basic semaphores of traditional thread libraries.

A Thread Pitfall: Deadlocks

Locks are a handy tool to synchronize access to data, and using them properly is the key to safe shared data. Unfortunately, locks aren't without their dangers, especially when multiple locks are involved. Consider the following code:

```
use threads;

my $a : shared = 4;
my $b : shared = "foo";
my $thr1 = threads->new(sub {
    lock($a);
    threads->yield;
    sleep 20;
    lock($b);
});
my $thr2 = threads->new(sub {
    lock($b);
    threads->yield;
    sleep 20;
    lock($a);
});
```

This program will probably hang until you kill it. The only way it won't hang is if one of the two threads acquires both locks first. A guaranteed-to-hang version is more complicated, but the principle is the same.

The first thread will grab a lock on `$a`, then, after a pause during which the second thread has probably had time to do some work, try to grab a lock on `$b`. Meanwhile, the second thread grabs a lock on `$b`, then later tries to grab a lock on `$a`. The second lock attempt for both threads will block, each waiting for the other to release its lock.

This condition is called a deadlock, and it occurs whenever two or more threads are trying to get locks on resources that the others own. Each thread will block, waiting for the other to release a lock on a resource. That never happens, though, since the thread with the resource is itself waiting for a lock to be released.

There are a number of ways to handle this sort of problem. The best way is to always have all threads acquire locks in the exact same order. If, for example, you lock variables `$a`, `$b`, and `$c`, always lock `$a` before `$b`, and `$b` before `$c`. It's also best to hold on to locks for as short a period of time to minimize the risks of deadlock.

The other synchronization primitives described below can suffer from similar problems.

Queues: Passing Data Around

A queue is a special thread-safe object that lets you put data in one end and take it out the other without having to worry about synchronization issues. They're pretty straightforward, and look like this:

```
use threads;
use Thread::Queue;
```

```

my $DataQueue = Thread::Queue->new;
$thr = threads->new(sub {
    while ($DataElement = $DataQueue->dequeue) {
        print "Popped $DataElement off the queue\n";
    }
});

$DataQueue->enqueue(12);
$DataQueue->enqueue("A", "B", "C");
$DataQueue->enqueue(\$thr);
sleep 10;
$DataQueue->enqueue(undef);
$thr->join;

```

You create the queue with `new Thread::Queue`. Then you can add lists of scalars onto the end with `enqueue()`, and pop scalars off the front of it with `dequeue()`. A queue has no fixed size, and can grow as needed to hold everything pushed on to it.

If a queue is empty, `dequeue()` blocks until another thread enqueues something. This makes queues ideal for event loops and other communications between threads.

Semaphores: Synchronizing Data Access

Semaphores are a kind of generic locking mechanism. In their most basic form, they behave very much like lockable scalars, except that they can't hold data, and that they must be explicitly unlocked. In their advanced form, they act like a kind of counter, and can allow multiple threads to have the 'lock' at any one time.

Basic semaphores

Semaphores have two methods, `down()` and `up()`: `down()` decrements the resource count, while `up()` increments it. Calls to `down()` will block if the semaphore's current count would decrement below zero. This program gives a quick demonstration:

```

use threads qw(yield);
use Thread::Semaphore;

my $semaphore = new Thread::Semaphore;
my $GlobalVariable : shared = 0;

$thr1 = new threads \&sample_sub, 1;
$thr2 = new threads \&sample_sub, 2;
$thr3 = new threads \&sample_sub, 3;

sub sample_sub {
    my $SubNumber = shift @_;
    my $TryCount = 10;
    my $LocalCopy;
    sleep 1;
    while ($TryCount-->0) {
        $semaphore->down;
        $LocalCopy = $GlobalVariable;
        print "$TryCount tries left for sub $SubNumber (\$GlobalVariable is $Glob";
        yield;
        sleep 2;
        $LocalCopy++;
        $GlobalVariable = $LocalCopy;
        $semaphore->up;
    }
}

```

```
$thr1->join;
$thr2->join;
$thr3->join;
```

The three invocations of the subroutine all operate in sync. The semaphore, though, makes sure that only one thread is accessing the global variable at once.

Advanced Semaphores

By default, semaphores behave like locks, letting only one thread `down()` them at a time. However, there are other uses for semaphores.

Each semaphore has a counter attached to it. By default, semaphores are created with the counter set to one, `down()` decrements the counter by one, and `up()` increments by one. However, we can override any or all of these defaults simply by passing in different values:

```
use threads;
use Thread::Semaphore;
my $semaphore = Thread::Semaphore->new(5);
               # Creates a semaphore with the counter set to five

$thr1 = threads->new(\&sub1);
$thr2 = threads->new(\&sub1);

sub sub1 {
    $semaphore->down(5); # Decrements the counter by five
    # Do stuff here
    $semaphore->up(5);  # Increment the counter by five
}

$thr1->detach;
$thr2->detach;
```

If `down()` attempts to decrement the counter below zero, it blocks until the counter is large enough. Note that while a semaphore can be created with a starting count of zero, any `up()` or `down()` always changes the counter by at least one, and so `$semaphore->down(0)` is the same as `$semaphore->down(1)`.

The question, of course, is why would you do something like this? Why create a semaphore with a starting count that's not one, or why decrement/increment it by more than one? The answer is resource availability. Many resources that you want to manage access for can be safely used by more than one thread at once.

For example, let's take a GUI driven program. It has a semaphore that it uses to synchronize access to the display, so only one thread is ever drawing at once. Handy, but of course you don't want any thread to start drawing until things are properly set up. In this case, you can create a semaphore with a counter set to zero, and up it when things are ready for drawing.

Semaphores with counters greater than one are also useful for establishing quotas. Say, for example, that you have a number of threads that can do I/O at once. You don't want all the threads reading or writing at once though, since that can potentially swamp your I/O channels, or deplete your process' quota of filehandles. You can use a semaphore initialized to the number of concurrent I/O requests (or open files) that you want at any one time, and have your threads quietly block and unblock themselves.

Larger increments or decrements are handy in those cases where a thread needs to check out or return a number of resources at once.

`cond_wait()` and `cond_signal()`

These two functions can be used in conjunction with locks to notify co-operating threads that a resource has become available. They are very similar in use to the functions found in `pthread`s. However for most purposes, queues are simpler to use and more intuitive. See [threads::shared](#) for more details.

General Thread Utility Routines

We've covered the workhorse parts of Perl's threading package, and with these tools you should be well on your way to writing threaded code and packages. There are a few useful little pieces that didn't really fit in anywhere else.

What Thread Am I In?

The `< threads-self` class method provides your program with a way to get an object representing the thread it's currently in. You can use this object in the same way as the ones returned from thread creation.

Thread IDs

`tid()` is a thread object method that returns the thread ID of the thread the object represents. Thread IDs are integers, with the main thread in a program being 0. Currently Perl assigns a unique tid to every thread ever created in your program, assigning the first thread to be created a tid of 1, and increasing the tid by 1 for each new thread that's created.

Are These Threads The Same?

The `equal()` method takes two thread objects and returns true if the objects represent the same thread, and false if they don't.

Thread objects also have an overloaded `==` comparison so that you can do comparison on them as you would with normal objects.

What Threads Are Running?

`< threads-list` returns a list of thread objects, one for each thread that's currently running and not detached. Handy for a number of things, including cleaning up at the end of your program:

```
# Loop through all the threads
foreach $thr (threads->list) {
    # Don't join the main thread or ourselves
    if ($thr->tid && !threads::equal($thr, threads->self)) {
        $thr->join;
    }
}
```

If some threads have not finished running when the main Perl thread ends, Perl will warn you about it and die, since it is impossible for Perl to clean up itself while other threads are running

A Complete Example

Confused yet? It's time for an example program to show some of the things we've covered. This program finds prime numbers using threads.

```
1  #!/usr/bin/perl -w
2  # prime-pthread, courtesy of Tom Christiansen
3
4  use strict;
5
6  use threads;
7  use Thread::Queue;
8
9  my $stream = new Thread::Queue;
10 my $kid     = new threads(\&check_num, $stream, 2);
11
12 for my $i ( 3 .. 1000 ) {
13     $stream->enqueue($i);
14 }
15
16 $stream->enqueue(undef);
```



```

17 $kid->join;
18
19 sub check_num {
20     my ($upstream, $cur_prime) = @_;
21     my $kid;
22     my $downstream = new Thread::Queue;
23     while (my $num = $upstream->dequeue) {
24         next unless $num % $cur_prime;
25         if ($kid) {
26             $downstream->enqueue($num);
27         } else {
28             print "Found prime $num\n";
29             $kid = new threads(\&check_num, $downstream, $num);
30         }
31     }
32     $downstream->enqueue(undef) if $kid;
33     $kid->join    if $kid;
34 }

```

This program uses the pipeline model to generate prime numbers. Each thread in the pipeline has an input queue that feeds numbers to be checked, a prime number that it's responsible for, and an output queue into which it funnels numbers that have failed the check. If the thread has a number that's failed its check and there's no child thread, then the thread must have found a new prime number. In that case, a new child thread is created for that prime and stuck on the end of the pipeline.

This probably sounds a bit more confusing than it really is, so let's go through this program piece by piece and see what it does. (For those of you who might be trying to remember exactly what a prime number is, it's a number that's only evenly divisible by itself and 1)

The bulk of the work is done by the `check_num()` subroutine, which takes a reference to its input queue and a prime number that it's responsible for. After pulling in the input queue and the prime that the subroutine's checking (line 20), we create a new queue (line 22) and reserve a scalar for the thread that we're likely to create later (line 21).

The while loop from lines 23 to line 31 grabs a scalar off the input queue and checks against the prime this thread is responsible for. Line 24 checks to see if there's a remainder when we modulo the number to be checked against our prime. If there is one, the number must not be evenly divisible by our prime, so we need to either pass it on to the next thread if we've created one (line 26) or create a new thread if we haven't.

The new thread creation is line 29. We pass on to it a reference to the queue we've created, and the prime number we've found.

Finally, once the loop terminates (because we got a 0 or undef in the queue, which serves as a note to die), we pass on the notice to our child and wait for it to exit if we've created a child (lines 32 and 37).

Meanwhile, back in the main thread, we create a queue (line 9) and the initial child thread (line 10), and pre-seed it with the first prime: 2. Then we queue all the numbers from 3 to 1000 for checking (lines 12–14), then queue a die notice (line 16) and wait for the first child thread to terminate (line 17). Because a child won't die until its child has died, we know that we're done once we return from the join.

That's how it works. It's pretty simple; as with many Perl programs, the explanation is much longer than the program.

Performance considerations

The main thing to bear in mind when comparing ithreads to other threading models is the fact that for each new thread created, a complete copy of all the variables and data of the parent thread has to be taken. Thus thread creation can be quite expensive, both in terms of memory usage and time spent in creation. The ideal way to reduce these costs is to have a relatively short number of long-lived threads, all created fairly early on – before the base thread has accumulated too much data. Of course, this may not always be possible, so

compromises have to be made. However, after a thread has been created, its performance and extra memory usage should be little different than ordinary code.

Also note that under the current implementation, shared variables use a little more memory and are a little slower than ordinary variables.

Process-scope Changes

Note that while threads themselves are separate execution threads and Perl data is thread-private unless explicitly shared, the threads can affect process-scope state, affecting all the threads.

The most common example of this is changing the current working directory using `chdir()`. One thread calls `chdir()`, and the working directory of all the threads changes.

Even more drastic example of a process-scope change is `chroot()`: the root directory of all the threads changes, and no thread can undo it (as opposed to `chdir()`).

Further examples of process-scope changes include `umask()` and changing `uids/gids`.

Thinking of mixing `fork()` and threads? Please lie down and wait until the feeling passes— but in case you really want to know, the semantics is that `fork()` duplicates all the threads. (In UNIX, at least, other platforms will do something different.)

Similarly, mixing signals and threads should not be attempted. Implementations are platform-dependent, and even the POSIX semantics may not be what you expect (and Perl doesn't even give you the full POSIX API).

Thread-Safety of System Libraries

Whether various library calls are thread-safe is outside the control of Perl. Calls often suffering from not being thread-safe include: `localtime()`, `gmtime()`, `get{gr,host,net,proto,serv,pw}*()`, `readdir()`, `rand()`, and `srand()` — in general, calls that depend on some global external state.

If the system Perl is compiled in has thread-safe variants of such calls, they will be used. Beyond that, Perl is at the mercy of the thread-safety or -unsafety of the calls. Please consult your C library call documentation.

In some platforms the thread-safe interfaces may fail if the result buffer is too small (for example `getgrent()` may return quite large group member lists). Perl will retry growing the result buffer a few times, but only up to 64k (for safety reasons).

Conclusion

A complete thread tutorial could fill a book (and has, many times), but with what we've covered in this introduction, you should be well on your way to becoming a threaded Perl expert.

Bibliography

Here's a short bibliography courtesy of Jürgen Christoffel:

Introductory Texts

Birrell, Andrew D. An Introduction to Programming with Threads. Digital Equipment Corporation, 1989, DEC-SRC Research Report #35 online as <http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-035.html> (highly recommended)

Robbins, Kay. A., and Steven Robbins. Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading. Prentice-Hall, 1996.

Lewis, Bill, and Daniel J. Berg. Multithreaded Programming with Pthreads. Prentice Hall, 1997, ISBN 0-13-443698-9 (a well-written introduction to threads).

Nelson, Greg (editor). Systems Programming with Modula-3. Prentice Hall, 1991, ISBN 0-13-590464-1.

Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. Pthreads Programming. O'Reilly & Associates, 1996, ISBN 156592-115-1 (covers POSIX threads).

OS-Related References

Boykin, Joseph, David Kirschen, Alan Langerman, and Susan LoVerso. Programming under Mach. Addison-Wesley, 1994, ISBN 0-201-52739-1.

Tanenbaum, Andrew S. Distributed Operating Systems. Prentice Hall, 1995, ISBN 0-13-219908-4 (great textbook).

Silberschatz, Abraham, and Peter B. Galvin. Operating System Concepts, 4th ed. Addison-Wesley, 1995, ISBN 0-201-59292-4

Other References

Arnold, Ken and James Gosling. The Java Programming Language, 2nd ed. Addison-Wesley, 1998, ISBN 0-201-31006-6.

comp.programming.threads FAQ, <http://www.serpentine.com/~bos/threads-faq/>

Le Sergent, T. and B. Berthomieu. "Incremental MultiThreaded Garbage Collection on Virtually Shared Memory Architectures" in Memory Management: Proc. of the International Workshop IWMM 92, St. Malo, France, September 1992, Yves Bekkers and Jacques Cohen, eds. Springer, 1992, ISBN 3540-55940-X (real-life thread applications).

Artur Bergman, "Where Wizards Fear To Tread", June 11, 2002,
<http://www.perl.com/pub/a/2002/06/11/threads.html>

Acknowledgements

Thanks (in no particular order) to Chaim Frenkel, Steve Fink, Gurusamy Sarathy, Ilya Zakharevich, Benjamin Sugars, Jürgen Christoffel, Joshua Pritikin, and Alan Burlison, for their help in reality-checking and polishing this article. Big thanks to Tom Christiansen for his rewrite of the prime number generator.

AUTHOR

Dan Sugalski <dan@sidhe.org>

Slightly modified by Arthur Bergman to fit the new thread model/module.

Reworked slightly by Jörg Walter <jwalt@cpan.org> to be more concise about thread-safety of perl code.

Copyrights

The original version of this article originally appeared in The Perl Journal #10, and is copyright 1998 The Perl Journal. It appears courtesy of Jon Orwant and The Perl Journal. This document may be distributed under the same terms as Perl itself.

For more information please see [threads](#) and [threads::shared](#).

NAME

perltie – how to hide an object class in a simple variable

SYNOPSIS

```
tie VARIABLE, CLASSNAME, LIST
$object = tied VARIABLE
untie VARIABLE
```

DESCRIPTION

Prior to release 5.0 of Perl, a programmer could use `dbmopen()` to connect an on-disk database in the standard Unix `dbm(3x)` format magically to a `%HASH` in their program. However, their Perl was either built with one particular `dbm` library or another, but not both, and you couldn't extend this mechanism to other packages or types of variables.

Now you can.

The `tie()` function binds a variable to a class (package) that will provide the implementation for access methods for that variable. Once this magic has been performed, accessing a tied variable automatically triggers method calls in the proper class. The complexity of the class is hidden behind magic methods calls. The method names are in ALL CAPS, which is a convention that Perl uses to indicate that they're called implicitly rather than explicitly—just like the `BEGIN()` and `END()` functions.

In the `tie()` call, `VARIABLE` is the name of the variable to be enchanted. `CLASSNAME` is the name of a class implementing objects of the correct type. Any additional arguments in the `LIST` are passed to the appropriate constructor method for that class—meaning `TIESCALAR()`, `TIEARRAY()`, `TIEHASH()`, or `TIEHANDLE()`. (Typically these are arguments such as might be passed to the `dbmopen()` function of C.) The object returned by the "new" method is also returned by the `tie()` function, which would be useful if you wanted to access other methods in `CLASSNAME`. (You don't actually have to return a reference to a right "type" (e.g., `HASH` or `CLASSNAME`) so long as it's a properly blessed object.) You can also retrieve a reference to the underlying object using the `tied()` function.

Unlike `dbmopen()`, the `tie()` function will not use or require a module for you—you need to do that explicitly yourself.

Tying Scalars

A class implementing a tied scalar should define the following methods: `TIESCALAR`, `FETCH`, `STORE`, and possibly `UNTIE` and/or `DESTROY`.

Let's look at each in turn, using as an example a tie class for scalars that allows the user to do something like:

```
tie $his_speed, 'Nice', getppid();
tie $my_speed, 'Nice', $$;
```

And now whenever either of those variables is accessed, its current system priority is retrieved and returned. If those variables are set, then the process's priority is changed!

We'll use Jarkko Hietaniemi <jhi@iki.fi>'s `BSD::Resource` class (not included) to access the `PRIO_PROCESS`, `PRIO_MIN`, and `PRIO_MAX` constants from your system, as well as the `getpriority()` and `setpriority()` system calls. Here's the preamble of the class.

```
package Nice;
use Carp;
use BSD::Resource;
use strict;
$Nice::DEBUG = 0 unless defined $Nice::DEBUG;
```

TIESCALAR classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference to a new scalar (probably anonymous) that it's creating. For example:

```
sub TIESCALAR {
    my $class = shift;
    my $pid = shift || $$; # 0 means me

    if ($pid !~ /\d+$/) {
        carp "Nice::Tie::Scalar got non-numeric pid $pid" if $^W;
        return undef;
    }

    unless (kill 0, $pid) { # EPERM or ERSCH, no doubt
        carp "Nice::Tie::Scalar got bad pid $pid: $!" if $^W;
        return undef;
    }

    return bless \$pid, $class;
}
```

This tie class has chosen to return an error rather than raising an exception if its constructor should fail. While this is how `dbmopen()` works, other classes may well not wish to be so forgiving. It checks the global variable `$^W` to see whether to emit a bit of noise anyway.

FETCH this

This method will be triggered every time the tied variable is accessed (read). It takes no arguments beyond its self reference, which is the object representing the scalar we're dealing with. Because in this case we're using just a **SCALAR** ref for the tied scalar object, a simple `$$self` allows the method to get at the real value stored there. In our example below, that real value is the process ID to which we've tied our variable.

```
sub FETCH {
    my $self = shift;
    confess "wrong type" unless ref $self;
    croak "usage error" if @_;
    my $nicety;
    local($!) = 0;
    $nicety = getpriority(PRIO_PROCESS, $$self);
    if ($!) { croak "getpriority failed: $!" }
    return $nicety;
}
```

This time we've decided to blow up (raise an exception) if the renice fails—there's no place for us to return an error otherwise, and it's probably the right thing to do.

STORE this, value

This method will be triggered every time the tied variable is set (assigned). Beyond its self reference, it also expects one (and only one) argument—the new value the user is trying to assign.

```
sub STORE {
    my $self = shift;
    confess "wrong type" unless ref $self;
    my $new_nicety = shift;
    croak "usage error" if @_;

    if ($new_nicety < PRIO_MIN) {
        carp sprintf
            "WARNING: priority %d less than minimum system priority %d",

```

```

        $new_nicety, PRIO_MIN if $^W;
        $new_nicety = PRIO_MIN;
    }

    if ($new_nicety > PRIO_MAX) {
        carp sprintf
            "WARNING: priority %d greater than maximum system priority %d",
            $new_nicety, PRIO_MAX if $^W;
        $new_nicety = PRIO_MAX;
    }

    unless (defined setpriority(PRIO_PROCESS, $$self, $new_nicety)) {
        confess "setpriority failed: $!";
    }

    return $new_nicety;
}

```

UNTIE this

This method will be triggered when the `untie` occurs. This can be useful if the class needs to know when no further calls will be made. (Except `DESTROY` of course.) See [The untie Gotcha](#) below for more details.

DESTROY this

This method will be triggered when the tied variable needs to be destructed. As with other object classes, such a method is seldom necessary, because Perl deallocates its moribund object's memory for you automatically—this isn't C++, you know. We'll use a `DESTROY` method here for debugging purposes only.

```

sub DESTROY {
    my $self = shift;
    confess "wrong type" unless ref $self;
    carp "[ Nice::DESTROY pid $$self ]" if $Nice::DEBUG;
}

```

That's about all there is to it. Actually, it's more than all there is to it, because we've done a few nice things here for the sake of completeness, robustness, and general aesthetics. Simpler `TIESCALAR` classes are certainly possible.

Tying Arrays

A class implementing a tied ordinary array should define the following methods: `TIEARRAY`, `FETCH`, `STORE`, `FETCHSIZE`, `STORESIZE` and perhaps `UNTIE` and/or `DESTROY`.

`FETCHSIZE` and `STORESIZE` are used to provide `$#array` and equivalent scalar(`@array`) access.

The methods `POP`, `PUSH`, `SHIFT`, `UNSHIFT`, `SPLICE`, `DELETE`, and `EXISTS` are required if the perl operator with the corresponding (but lowercase) name is to operate on the tied array. The `Tie::Array` class can be used as a base class to implement the first five of these in terms of the basic methods above. The default implementations of `DELETE` and `EXISTS` in `Tie::Array` simply croak.

In addition `EXTEND` will be called when perl would have pre-extended allocation in a real array.

For this discussion, we'll implement an array whose elements are a fixed size at creation. If you try to create an element larger than the fixed size, you'll take an exception. For example:

```

use FixedElem_Array;
tie @array, 'FixedElem_Array', 3;
$array[0] = 'cat'; # ok.
$array[1] = 'dogs'; # exception, length('dogs') > 3.

```

The preamble code for the class is as follows:

```
package FixedElem_Array;
use Carp;
use strict;
```

TIEARRAY classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference through which the new array (probably an anonymous ARRAY ref) will be accessed.

In our example, just to show you that you don't *really* have to return an ARRAY reference, we'll choose a HASH reference to represent our object. A HASH works out well as a generic record type: the {ELEM_SIZE} field will store the maximum element size allowed, and the {ARRAY} field will hold the true ARRAY ref. If someone outside the class tries to dereference the object returned (doubtless thinking it an ARRAY ref), they'll blow up. This just goes to show you that you should respect an object's privacy.

```
sub TIEARRAY {
    my $class = shift;
    my $elemsize = shift;
    if ( @_ || $elemsize =~ /\D/ ) {
        croak "usage: tie ARRAY, ' ' . __PACKAGE__ . ' ', elem_size";
    }
    return bless {
        ELEM_SIZE => $elemsize,
        ARRAY => [],
    }, $class;
}
```

FETCH this, index

This method will be triggered every time an individual element the tied array is accessed (read). It takes one argument beyond its self reference: the index whose value we're trying to fetch.

```
sub FETCH {
    my $self = shift;
    my $index = shift;
    return $self->{ARRAY}->[$index];
}
```

If a negative array index is used to read from an array, the index will be translated to a positive one internally by calling FETCHSIZE before being passed to FETCH.

As you may have noticed, the name of the FETCH method (et al.) is the same for all accesses, even though the constructors differ in names (TIESCALAR vs TIEARRAY). While in theory you could have the same class servicing several tied types, in practice this becomes cumbersome, and it's easiest to keep them at simply one tie type per class.

STORE this, index, value

This method will be triggered every time an element in the tied array is set (written). It takes two arguments beyond its self reference: the index at which we're trying to store something and the value we're trying to put there.

In our example, undef is really \$self->{ELEM_SIZE} number of spaces so we have a little more work to do here:

```
sub STORE {
    my $self = shift;
    my ( $index, $value ) = @_;
    if ( length $value > $self->{ELEM_SIZE} ) {
        croak "length of $value is greater than $self->{ELEM_SIZE}";
    }
}
```

```

    # fill in the blanks
    $self->EXTEND( $index ) if $index > $self->FETCHSIZE();
    # right justify to keep element size for smaller elements
    $self->{ARRAY}->[$index] = sprintf "%$self->{ELEMSIZE}s", $value;
}

```

Negative indexes are treated the same as with FETCH.

FETCHSIZE this

Returns the total number of items in the tied array associated with object *this*. (Equivalent to `scalar(@array)`). For example:

```

sub FETCHSIZE {
    my $self = shift;
    return scalar @{$self->{ARRAY}};
}

```

STORESIZE this, count

Sets the total number of items in the tied array associated with object *this* to be *count*. If this makes the array larger then class's mapping of undef should be returned for new positions. If the array becomes smaller then entries beyond count should be deleted.

In our example, 'undef' is really an element containing `$self->{ELEMSIZE}` number of spaces. Observe:

```

sub STORESIZE {
    my $self = shift;
    my $count = shift;
    if ( $count > $self->FETCHSIZE() ) {
        foreach ( $count - $self->FETCHSIZE() .. $count ) {
            $self->STORE( $_, ' ' );
        }
    } elsif ( $count < $self->FETCHSIZE() ) {
        foreach ( 0 .. $self->FETCHSIZE() - $count - 2 ) {
            $self->POP();
        }
    }
}

```

EXTEND this, count

Informative call that array is likely to grow to have *count* entries. Can be used to optimize allocation. This method need do nothing.

In our example, we want to make sure there are no blank (undef) entries, so EXTEND will make use of STORESIZE to fill elements as needed:

```

sub EXTEND {
    my $self = shift;
    my $count = shift;
    $self->STORESIZE( $count );
}

```

EXISTS this, key

Verify that the element at index *key* exists in the tied array *this*.

In our example, we will determine that if an element consists of `$self->{ELEMSIZE}` spaces only, it does not exist:

```

sub EXISTS {
    my $self = shift;

```



```

    my $index = shift;
    return 0 if ! defined $self->{ARRAY}->[$index] ||
        $self->{ARRAY}->[$index] eq ' ' x $self->{ELEMSIZE};
    return 1;
}

```

DELETE this, key

Delete the element at index *key* from the tied array *this*.

In our example, a deleted item is `$self-{ELEMSIZE}` spaces:

```

sub DELETE {
    my $self = shift;
    my $index = shift;
    return $self->STORE( $index, ' ' );
}

```

CLEAR this

Clear (remove, delete, ...) all values from the tied array associated with object *this*. For example:

```

sub CLEAR {
    my $self = shift;
    return $self->{ARRAY} = [];
}

```

PUSH this, LIST

Append elements of *LIST* to the array. For example:

```

sub PUSH {
    my $self = shift;
    my @list = @_;
    my $last = $self->FETCHSIZE();
    $self->STORE( $last + $_, $list[$_] ) foreach 0 .. $#list;
    return $self->FETCHSIZE();
}

```

POP this

Remove last element of the array and return it. For example:

```

sub POP {
    my $self = shift;
    return pop @{$self->{ARRAY}};
}

```

SHIFT this

Remove the first element of the array (shifting other elements down) and return it. For example:

```

sub SHIFT {
    my $self = shift;
    return shift @{$self->{ARRAY}};
}

```

UNSHIFT this, LIST

Insert *LIST* elements at the beginning of the array, moving existing elements up to make room. For example:

```

sub UNSHIFT {
    my $self = shift;
    my @list = @_;
    my $size = scalar( @list );

```

```

    # make room for our list
    @{$self->{ARRAY}}[ $size .. $#{$self->{ARRAY}} + $size ]
        = @{$self->{ARRAY}};
    $self->STORE( $_, $list[$_] ) foreach 0 .. $#list;
}

```

SPLICE this, offset, length, LIST

Perform the equivalent of `splice` on the array.

offset is optional and defaults to zero, negative values count back from the end of the array.

length is optional and defaults to rest of the array.

LIST may be empty.

Returns a list of the original *length* elements at *offset*.

In our example, we'll use a little shortcut if there is a *LIST*:

```

sub SPLICE {
    my $self = shift;
    my $offset = shift || 0;
    my $length = shift || $self->FETCHSIZE() - $offset;
    my @list = ();
    if ( @_ ) {
        tie @list, __PACKAGE__, $self->{ELEMSIZE};
        @list = @_;
    }
    return splice @{$self->{ARRAY}}, $offset, $length, @list;
}

```

UNTIE this

Will be called when `untie` happens. (See [The untie Gotcha](#) below.)

DESTROY this

This method will be triggered when the tied variable needs to be destructed. As with the scalar tie class, this is almost never needed in a language that does its own garbage collection, so this time we'll just leave it out.

Tying Hashes

Hashes were the first Perl data type to be tied (see `dbmopen()`). A class implementing a tied hash should define the following methods: `TIEHASH` is the constructor. `FETCH` and `STORE` access the key and value pairs. `EXISTS` reports whether a key is present in the hash, and `DELETE` deletes one. `CLEAR` empties the hash by deleting all the key and value pairs. `FIRSTKEY` and `NEXTKEY` implement the `keys()` and `each()` functions to iterate over all the keys. `UNTIE` is called when `untie` happens, and `DESTROY` is called when the tied variable is garbage collected.

If this seems like a lot, then feel free to inherit from merely the standard `Tie::StdHash` module for most of your methods, redefining only the interesting ones. See [Tie::Hash](#) for details.

Remember that Perl distinguishes between a key not existing in the hash, and the key existing in the hash but having a corresponding value of `undef`. The two possibilities can be tested with the `exists()` and `defined()` functions.

Here's an example of a somewhat interesting tied hash class: it gives you a hash representing a particular user's dot files. You index into the hash with the name of the file (minus the dot) and you get back that dot file's contents. For example:

```

use DotFiles;
tie %dot, 'DotFiles';
if ( $dot{profile} =~ /MANPATH/ ||

```

```

        $dot{login}    =~ /MANPATH/ ||
        $dot{cshrc}    =~ /MANPATH/   )
    {
        print "you seem to set your MANPATH\n";
    }

```

Or here's another sample of using our tied class:

```

tie %him, 'DotFiles', 'daemon';
foreach $f ( keys %him ) {
    printf "daemon dot file %s is size %d\n",
        $f, length $him{$f};
}

```

In our tied hash DotFiles example, we use a regular hash for the object containing several important fields, of which only the `{LIST}` field will be what the user thinks of as the real hash.

USER

whose dot files this object represents

HOME

where those dot files live

CLOBBER

whether we should try to change or remove those dot files

LIST the hash of dot file names and content mappings

Here's the start of *Dotfiles.pm*:

```

package DotFiles;
use Carp;
sub whowasi { (caller(1))[3] . '()' }
my $DEBUG = 0;
sub debug { $DEBUG = @_ ? shift : 1 }

```

For our example, we want to be able to emit debugging info to help in tracing during development. We keep also one convenience function around internally to help print out warnings; `whowasi()` returns the function name that calls it.

Here are the methods for the DotFiles tied hash.

TIEHASH classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference through which the new object (probably but not necessarily an anonymous hash) will be accessed.

Here's the constructor:

```

sub TIEHASH {
    my $self = shift;
    my $user = shift || $>;
    my $dotdir = shift || '';
    croak "usage: @{$[&whowasi]} [USER [DOTDIR]]" if @_;
    $user = getpwuid($user) if $user =~ /^~\d+$/;
    my $dir = (getpwnam($user))[7]
        || croak "@{$[&whowasi]}: no user $user";
    $dir .= "/$dotdir" if $dotdir;

    my $node = {
        USER    => $user,
        HOME     => $dir,
    }
}

```

```

        LIST      => {},
        CLOBBER => 0,
    };

    opendir(DIR, $dir)
        || croak "@{&whowasi}: can't opendir $dir: $!";
    foreach $dot ( grep /^\.\/ && -f "$dir/$_", readdir(DIR) ) {
        $dot =~ s/^\.\/;
        $node->{LIST}{$dot} = undef;
    }
    closedir DIR;
    return bless $node, $self;
}

```

It's probably worth mentioning that if you're going to filetest the return values out of a `readdir`, you'd better prepend the directory in question. Otherwise, because we didn't `chdir()` there, it would have been testing the wrong file.

FETCH this, key

This method will be triggered every time an element in the tied hash is accessed (read). It takes one argument beyond its self reference: the key whose value we're trying to fetch.

Here's the fetch for our DotFiles example.

```

sub FETCH {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $dir = $self->{HOME};
    my $file = "$dir/.$dot";

    unless (exists $self->{LIST}->{$dot} || -f $file) {
        carp "@{&whowasi}: no $dot file" if $DEBUG;
        return undef;
    }

    if (defined $self->{LIST}->{$dot}) {
        return $self->{LIST}->{$dot};
    } else {
        return $self->{LIST}->{$dot} = `cat $dir/.$dot`;
    }
}

```

It was easy to write by having it call the Unix `cat(1)` command, but it would probably be more portable to open the file manually (and somewhat more efficient). Of course, because dot files are a Unixy concept, we're not that concerned.

STORE this, key, value

This method will be triggered every time an element in the tied hash is set (written). It takes two arguments beyond its self reference: the index at which we're trying to store something, and the value we're trying to put there.

Here in our DotFiles example, we'll be careful not to let them try to overwrite the file unless they've called the `clobber()` method on the original object reference returned by `tie()`.

```

sub STORE {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $value = shift;

```

```

my $file = $self->{HOME} . "/.$dot";
my $user = $self->{USER};

croak "@{[&whowasi]}: $file not clobberable"
    unless $self->{CLOBBER};

open(F, "> $file") || croak "can't open $file: $!";
print F $value;
close(F);
}

```

If they wanted to clobber something, they might say:

```

$ob = tie %daemon_dots, 'daemon';
$ob->clobber(1);
$daemon_dots{signature} = "A true daemon\n";

```

Another way to lay hands on a reference to the underlying object is to use the `tied()` function, so they might alternately have set `clobber` using:

```

tie %daemon_dots, 'daemon';
tied(%daemon_dots)->clobber(1);

```

The `clobber` method is simply:

```

sub clobber {
    my $self = shift;
    $self->{CLOBBER} = @_ ? shift : 1;
}

```

DELETE this, key

This method is triggered when we remove an element from the hash, typically by using the `delete()` function. Again, we'll be careful to check whether they really want to clobber files.

```

sub DELETE {
    carp &whowasi if $DEBUG;

    my $self = shift;
    my $dot = shift;
    my $file = $self->{HOME} . "/.$dot";
    croak "@{[&whowasi]}: won't remove file $file"
        unless $self->{CLOBBER};
    delete $self->{LIST}->{$dot};
    my $success = unlink($file);
    carp "@{[&whowasi]}: can't unlink $file: $!" unless $success;
    $success;
}

```

The value returned by `DELETE` becomes the return value of the call to `delete()`. If you want to emulate the normal behavior of `delete()`, you should return whatever `FETCH` would have returned for this key. In this example, we have chosen instead to return a value which tells the caller whether the file was successfully deleted.

CLEAR this

This method is triggered when the whole hash is to be cleared, usually by assigning the empty list to it.

In our example, that would remove all the user's dot files! It's such a dangerous thing that they'll have to set `CLOBBER` to something higher than 1 to make it happen.

```

sub CLEAR {
    carp &whowasi if $DEBUG;
}

```

```

        my $self = shift;
        croak "@{[&whowasi]}: won't remove all dot files for $self->{USER}"
            unless $self->{CLOBBER} > 1;
        my $dot;
        foreach $dot ( keys %{$self->{LIST}} ) {
            $self->DELETE($dot);
        }
    }
}

```

EXISTS this, key

This method is triggered when the user uses the `exists()` function on a particular hash. In our example, we'll look at the `{LIST}` hash element for this:

```

sub EXISTS {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    return exists $self->{LIST}->{$dot};
}

```

FIRSTKEY this

This method will be triggered when the user is going to iterate through the hash, such as via a `keys()` or `each()` call.

```

sub FIRSTKEY {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $a = keys %{$self->{LIST}};           # reset each() iterator
    each %{$self->{LIST}}
}

```

NEXTKEY this, lastkey

This method gets triggered during a `keys()` or `each()` iteration. It has a second argument which is the last key that had been accessed. This is useful if you're carrying about ordering or calling the iterator from more than one sequence, or not really storing things in a hash anywhere.

For our example, we're using a real hash so we'll do just the simple thing, but we'll have to go through the `LIST` field indirectly.

```

sub NEXTKEY {
    carp &whowasi if $DEBUG;
    my $self = shift;
    return each %{$self->{LIST}}
}

```

UNTIE this

This is called when `untie` occurs. See [The untie Gotcha](#) below.

DESTROY this

This method is triggered when a tied hash is about to go out of scope. You don't really need it unless you're trying to add debugging or have auxiliary state to clean up. Here's a very simple function:

```

sub DESTROY {
    carp &whowasi if $DEBUG;
}

```

Note that functions such as `keys()` and `values()` may return huge lists when used on large objects, like DBM files. You may prefer to use the `each()` function to iterate over such. Example:

```
# print out history file offsets
use NDBM_File;
tie(%HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

Tying FileHandles

This is partially implemented now.

A class implementing a tied filehandle should define the following methods: TIEHANDLE, at least one of PRINT, PRINTF, WRITE, READLINE, GETC, READ, and possibly CLOSE, UNTIE and DESTROY. The class can also provide: BINMODE, OPEN, EOF, FILENO, SEEK, TELL – if the corresponding perl operators are used on the handle.

It is especially useful when perl is embedded in some other program, where output to STDOUT and STDERR may have to be redirected in some special way. See nvi and the Apache module for examples.

In our example we're going to create a shouting handle.

```
package Shout;
```

TIEHANDLE classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference of some sort. The reference can be used to hold some internal information.

```
sub TIEHANDLE { print "<shout>\n"; my $i; bless \$i, shift }
```

WRITE this, LIST

This method will be called when the handle is written to via the syswrite function.

```
sub WRITE {
    $r = shift;
    my ($buf,$len,$offset) = @_;
    print "WRITE called, \$buf=$buf, \$len=$len, \$offset=$offset";
}
```

PRINT this, LIST

This method will be triggered every time the tied handle is printed to with the print() function. Beyond its self reference it also expects the list that was passed to the print function.

```
sub PRINT { $r = shift; $$r++; print join($,,map(uc($_),@_)), $\ }
```

PRINTF this, LIST

This method will be triggered every time the tied handle is printed to with the printf() function. Beyond its self reference it also expects the format and list that was passed to the printf function.

```
sub PRINTF {
    shift;
    my $fmt = shift;
    print sprintf($fmt, @_)." \n";
}
```

READ this, LIST

This method will be called when the handle is read from via the read or sysread functions.

```
sub READ {
    my $self = shift;
    my $bufref = $_[0];
    my (undef,$len,$offset) = @_;
```

```

        print "READ called, \$_buf=$_bufref, \$_len=$_len, \$_offset=$_offset";
        # add to $_bufref, set $_len to number of characters read
        $_len;
    }

```

READLINE this

This method will be called when the handle is read from via `<HANDLE`. The method should return `undef` when there is no more data.

```
sub READLINE { $r = shift; "READLINE called $$r times\n"; }
```

GETC this

This method will be called when the `getc` function is called.

```
sub GETC { print "Don't GETC, Get Perl"; return "a"; }
```

CLOSE this

This method will be called when the handle is closed via the `close` function.

```
sub CLOSE { print "CLOSE called.\n" }
```

UNTIE this

As with the other types of ties, this method will be called when `untie` happens. It may be appropriate to "auto CLOSE" when this occurs. See [The *untie* Gotcha](#) below.

DESTROY this

As with the other types of ties, this method will be called when the tied handle is about to be destroyed. This is useful for debugging and possibly cleaning up.

```
sub DESTROY { print "</shout>\n" }
```

Here's how to use our little example:

```

tie(*FOO, 'Shout');
print FOO "hello\n";
$a = 4; $b = 6;
print FOO $a, " plus ", $b, " equals ", $a + $b, "\n";
print <FOO>;

```

UNTIE this

You can define for all tie types an `UNTIE` method that will be called at `untie()`. See [The *untie* Gotcha](#) below.

The *untie* Gotcha

If you intend making use of the object returned from either `tie()` or `tied()`, and if the tie's target class defines a destructor, there is a subtle gotcha you *must* guard against.

As setup, consider this (admittedly rather contrived) example of a tie; all it does is use a file to keep a log of the values assigned to a scalar.

```

package Remember;

use strict;
use warnings;
use IO::File;

sub TIESCALAR {
    my $class = shift;
    my $filename = shift;
    my $handle = new IO::File "> $filename"
                      or die "Cannot open $filename: $!\n";

```



```

        print $handle "The Start\n";
        bless {FH => $handle, Value => 0}, $class;
    }

    sub FETCH {
        my $self = shift;
        return $self->{Value};
    }

    sub STORE {
        my $self = shift;
        my $value = shift;
        my $handle = $self->{FH};
        print $handle "$value\n";
        $self->{Value} = $value;
    }

    sub DESTROY {
        my $self = shift;
        my $handle = $self->{FH};
        print $handle "The End\n";
        close $handle;
    }

    1;

```

Here is an example that makes use of this tie:

```

use strict;
use Remember;

my $fred;
tie $fred, 'Remember', 'myfile.txt';
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat myfile.txt";

```

This is the output when it is executed:

```

The Start
1
4
5
The End

```

So far so good. Those of you who have been paying attention will have spotted that the tied object hasn't been used so far. So let's add an extra method to the Remember class to allow comments to be included in the file — say, something like this:

```

sub comment {
    my $self = shift;
    my $text = shift;
    my $handle = $self->{FH};
    print $handle $text, "\n";
}

```

And here is the previous example modified to use the comment method (which requires the tied object):

```

use strict;
use Remember;

my ($fred, $x);
$x = tie $fred, 'Remember', 'myfile.txt';
$fred = 1;
$fred = 4;
comment $x "changing...";
$fred = 5;
untie $fred;
system "cat myfile.txt";

```

When this code is executed there is no output. Here's why:

When a variable is tied, it is associated with the object which is the return value of the TIESCALAR, TIEARRAY, or TIEHASH function. This object normally has only one reference, namely, the implicit reference from the tied variable. When `untie()` is called, that reference is destroyed. Then, as in the first example above, the object's destructor (DESTROY) is called, which is normal for objects that have no more valid references; and thus the file is closed.

In the second example, however, we have stored another reference to the tied object in `$x`. That means that when `untie()` gets called there will still be a valid reference to the object in existence, so the destructor is not called at that time, and thus the file is not closed. The reason there is no output is because the file buffers have not been flushed to disk.

Now that you know what the problem is, what can you do to avoid it? Prior to the introduction of the optional UNTIE method the only way was the good old `-w` flag. Which will spot any instances where you call `untie()` and there are still valid references to the tied object. If the second script above this near the top use `warnings 'untie'` or was run with the `-w` flag, Perl prints this warning message:

```

untie attempted while 1 inner references still exist

```

To get the script to work properly and silence the warning make sure there are no valid references to the tied object *before* `untie()` is called:

```

undef $x;
untie $fred;

```

Now that UNTIE exists the class designer can decide which parts of the class functionality are really associated with `untie` and which with the object being destroyed. What makes sense for a given class depends on whether the inner references are being kept so that non-tie-related methods can be called on the object. But in most cases it probably makes sense to move the functionality that would have been in DESTROY to the UNTIE method.

If the UNTIE method exists then the warning above does not occur. Instead the UNTIE method is passed the count of "extra" references and can issue its own warning if appropriate. e.g. to replicate the no UNTIE case this method can be used:

```

sub UNTIE
{
    my ($obj,$count) = @_;
    carp "untie attempted while $count inner references still exist" if $count;
}

```

SEE ALSO

See [DB_File](#) or [Config](#) for some interesting `tie()` implementations. A good starting point for many `tie()` implementations is with one of the modules [Tie::Scalar](#), [Tie::Array](#), [Tie::Hash](#), or [Tie::Handle](#).

BUGS

You cannot easily tie a multilevel data structure (such as a hash of hashes) to a dbm file. The first problem is that all but GDBM and Berkeley DB have size limitations, but beyond that, you also have problems with how references are to be represented on disk. One experimental module that does attempt to address this need partially is the MLDBM module. Check your nearest CPAN site as described in [perlmodlib](#) for source code to MLDBM.

Tied filehandles are still incomplete. `sysopen()`, `truncate()`, `flock()`, `fcntl()`, `stat()` and `-X` can't currently be trapped.

AUTHOR

Tom Christiansen

TIEHANDLE by Sven Verdoolaege <skimo@dns.ufsia.ac.be> and Doug MacEachern <doug@osf.org>

UNTIE by Nick Ing-Simmons <nick@ing-simmons.net>

Tying Arrays by Casey West <casey@geeknest.com>

NAME

perltoc – perl documentation table of contents

DESCRIPTION

This page provides a brief table of contents for the rest of the Perl documentation set. It is meant to be scanned quickly or grepped through to locate the proper section you're looking for.

BASIC DOCUMENTATION**perl – Practical Extraction and Report Language**

SYNOPSIS

- Overview
- Tutorials
- Reference Manual
- Internals and C Language Interface
- Miscellaneous
- Language-Specific
- Platform-Specific

DESCRIPTION

AVAILABILITY

ENVIRONMENT

AUTHOR

FILES

SEE ALSO

DIAGNOSTICS

BUGS

NOTES

perlintro — a brief introduction and overview of Perl

DESCRIPTION

- What is Perl?

- Running Perl programs

- Basic syntax overview

- Perl variable types

- Scalars, Arrays, Hashes

- Variable scoping

- Conditional and looping constructs

- if, while, for, foreach

- Builtin operators and functions

- Arithmetic, Numeric comparison, String comparison, Boolean logic, Miscellaneous

- Files and I/O

- Regular expressions

- Simple matching, Simple substitution, More complex regular expressions, Parentheses for capturing, Other regexp features

- Writing subroutines

- OO Perl

- Using Perl modules

AUTHOR

perlfreq – frequently asked questions about Perl (\$Date: 2002/03/11

21:32:23 \$)

DESCRIPTION

perlfac: Structural overview of the FAQ.
[perlfac1](#): General Questions About Perl
[perlfac2](#): Obtaining and Learning about Perl
[perlfac3](#): Programming Tools
[perlfac4](#): Data Manipulation
[perlfac5](#): Files and Formats
[perlfac6](#): Regular Expressions
[perlfac7](#): General Perl Language Issues
[perlfac8](#): System Interaction
[perlfac9](#): Networking

About the perlfac documents

- Where to get the perlfac
- How to contribute to the perlfac
- What will happen if you mail your Perl programming problems to the authors

Credits

Author and Copyright Information

- Bundled Distributions
- Disclaimer

Changes

1/November/2000, 23/May/99, 13/April/99, 7/January/99, 22/June/98, 24/April/97, 23/April/97, 25/March/97, 18/March/97, 17/March/97 Version, Initial Release: 11/March/97

perlbook – Perl book information**DESCRIPTION****perlsyn – Perl syntax****DESCRIPTION**

Declarations
Simple statements
Compound statements
Loop Control
For Loops
Foreach Loops
Basic BLOCKs and Switch Statements
Goto
PODs: Embedded Documentation
Plain Old Comments (Not!)

perldata – Perl data types**DESCRIPTION**

Variable names
Context
Scalar values
Scalar value constructors
List value constructors
Slices
Typeglobs and Filehandles

SEE ALSO

perlop – Perl operators and precedence

SYNOPSIS

DESCRIPTION

Terms and List Operators (Leftward)
 The Arrow Operator
 Auto-increment and Auto-decrement
 Exponentiation
 Symbolic Unary Operators
 Binding Operators
 Multiplicative Operators
 Additive Operators
 Shift Operators
 Named Unary Operators
 Relational Operators
 Equality Operators
 Bitwise And
 Bitwise Or and Exclusive Or
 C-style Logical And
 C-style Logical Or
 Range Operators
 Conditional Operator
 Assignment Operators
 Comma Operator
 List Operators (Rightward)
 Logical Not
 Logical And
 Logical or and Exclusive Or
 C Operators Missing From Perl
 unary &, unary *, (TYPE)

Quote and Quote-like Operators

Regexp Quote-Like Operators

?PATTERN?, m/PATTERN/cgimosx, /PATTERN/cgimosx, q/STRING/, `STRING`,
 qq/STRING/, "STRING", qr/STRING/imosx, qx/STRING/, 'STRING', qw/STRING/,
 s/PATTERN/REPLACEMENT/egimosx, tr/SEARCHLIST/REPLACEMENTLIST/cds,
 y/SEARCHLIST/REPLACEMENTLIST/cds, <<EOF

Gory details of parsing quoted constructs

Finding the end, Removal of backslashes before delimiters, Interpolation, <<'EOF', m'',
 s'', tr///, y///, '', q//, "", ``, qq//, qx//, < <file*glob, ?RE?, /RE/,
 m/RE/, s/RE/fooo/, Interpolation of regular expressions, Optimization of regular expressions

I/O Operators
 Constant Folding
 Bitwise String Operators
 Integer Arithmetic
 Floating-point Arithmetic
 Bigger Numbers

perlsub – Perl subroutines

SYNOPSIS

DESCRIPTION

Private Variables via `my()`
 Persistent Private Variables
 Temporary Values via `local()`
 Lvalue subroutines
 Lvalue subroutines are EXPERIMENTAL

Passing Symbol Table Entries (typeglobs)
 When to Still Use `local()`
 Pass by Reference
 Prototypes
 Constant Functions
 Overriding Built-in Functions
 Autoloading
 Subroutine Attributes

SEE ALSO

perlfunc – Perl builtin functions

DESCRIPTION

Perl Functions by Category

Functions for SCALARs or strings, Regular expressions and pattern matching, Numeric functions, Functions for real @ARRAYs, Functions for list data, Functions for real %HASHes, Input and output functions, Functions for fixed length data or records, Functions for filehandles, files, or directories, Keywords related to the control flow of your perl program, Keywords related to scoping, Miscellaneous functions, Functions for processes and process groups, Keywords related to perl modules, Keywords related to classes and object-orientedness, Low-level socket functions, System V interprocess communication functions, Fetching user and group info, Fetching network info, Time-related functions, Functions new in perl5, Functions obsoleted in perl5

Portability

Alphabetical Listing of Perl Functions

`-X` FILEHANDLE, `-X` EXPR, `-X`, `abs` VALUE, `abs`, `accept`
`NEWSOCKET`, `GENERICSOCKET`, `alarm` SECONDS, `alarm`, `atan2` Y,X, `bind`
`SOCKET`, NAME, `binmode` FILEHANDLE, `LAYER`, `binmode` FILEHANDLE, `bless`
`REF`, CLASSNAME, `bless` REF, `caller` EXPR, `caller`, `chdir` EXPR, `chmod` LIST, `chomp`
 VARIABLE, `chomp`(LIST), `chomp`, `chop` VARIABLE, `chop`(LIST), `chop`, `chown` LIST, `chr`
 NUMBER, `chr`, `chroot` FILENAME, `chroot`, `close` FILEHANDLE, `close`, `closedir`
`DIRHANDLE`, `connect` SOCKET, NAME, `continue` BLOCK, `cos` EXPR, `cos`, `crypt`
`PLAINTEXT`, SALT, `dbmclose` HASH, `dbmopen` HASH, DBNAME, MASK, `defined` EXPR,
`defined`, `delete` EXPR, `die` LIST, `do` BLOCK, `do` SUBROUTINE(LIST), `do` EXPR, `dump`
 LABEL, `dump`, `each` HASH, `eof` FILEHANDLE, `eof` (), `eof`, `eval` EXPR, `eval` BLOCK, `exec`
 LIST, `exec` PROGRAM LIST, `exists` EXPR, `exit` EXPR, `exp` EXPR, `exp`, `fcntl`
 FILEHANDLE, FUNCTION, SCALAR, `fileno` FILEHANDLE, `flock`
 FILEHANDLE, OPERATION, `fork`, `format`, `formline` PICTURE, LIST, `getc` FILEHANDLE,
`getc`, `getlogin`, `getpeername` SOCKET, `getpgrp` PID, `getppid`, `getpriority` WHICH, WHO,
`getpwnam` NAME, `getgrnam` NAME, `gethostbyname` NAME, `getnetbyname` NAME,
`getprotobyname` NAME, `getpwuid` UID, `getrgid` GID, `getservbyname` NAME, PROTO,
`gethostbyaddr` ADDR, ADDRTYPE, `getnetbyaddr` ADDR, ADDRTYPE, `getprotobyname`
 NUMBER, `getservbyport` PORT, PROTO, `getpwent`, `getgrent`, `gethostent`, `getnetent`, `getprotoent`,
`getservent`, `setpwent`, `setgrent`, `sethostent` STAYOPEN, `setnetent` STAYOPEN, `setprotoent`
 STAYOPEN, `setservent` STAYOPEN, `endpwent`, `endgrent`, `endhostent`, `endnetent`, `endprotoent`,
`endservent`, `getsockname` SOCKET, `getsockopt` SOCKET, LEVEL, OPTNAME, `glob` EXPR,
`glob`, `gmtime` EXPR, `goto` LABEL, `goto` EXPR, `goto` &NAME, `grep` BLOCK LIST, `grep`
 EXPR, LIST, `hex` EXPR, `hex`, `import`, `index` STR, SUBSTR, POSITION, `index` STR, SUBSTR, int
 EXPR, int, `ioctl` FILEHANDLE, FUNCTION, SCALAR, `join` EXPR, LIST, `keys` HASH, `kill`

SIGNAL, LIST, last LABEL, last, lc EXPR, lc, lcfirst EXPR, lcfirst, length EXPR, length, link
 OLDFILE,NEWFILE, listen SOCKET,QUEUESIZE, local EXPR, localtime EXPR, lock
 THING, log EXPR, log, lstat EXPR, lstat, m//, map BLOCK LIST, map EXPR,LIST, mkdir
 FILENAME,MASK, mkdir FILENAME, msgctl ID,CMD,ARG, msgget KEY,FLAGS, msgrcv
 ID,VAR,SIZE,TYPE,FLAGS, msgsnd ID,MSG,FLAGS, my EXPR, my TYPE EXPR, my
 EXPR : ATTRS, my TYPE EXPR : ATTRS, next LABEL, next, no Module VERSION LIST,
 no Module VERSION, no Module LIST, no Module, oct EXPR, oct, open
 FILEHANDLE,EXPR, open FILEHANDLE,MODE,EXPR, open
 FILEHANDLE,MODE,EXPR,LIST, open FILEHANDLE,MODE,REFERENCE, open
 FILEHANDLE, opendir DIRHANDLE,EXPR, ord EXPR, ord, our EXPR, our EXPR TYPE, our
 EXPR : ATTRS, our TYPE EXPR : ATTRS, pack TEMPLATE,LIST, package NAMESPACE,
 package, pipe READHANDLE,WRITEHANDLE, pop ARRAY, pop, pos SCALAR, pos, print
 FILEHANDLE LIST, print LIST, print, printf FILEHANDLE FORMAT, LIST, printf
 FORMAT, LIST, prototype FUNCTION, push ARRAY,LIST, q/STRING/, qq/STRING/,
 qr/STRING/, qx/STRING/, qw/STRING/, quotemeta EXPR, quotemeta, rand EXPR, rand, read
 FILEHANDLE,SCALAR,LENGTH,OFFSET, read FILEHANDLE,SCALAR,LENGTH, readdir
 DIRHANDLE, readline EXPR, readlink EXPR, readlink, readpipe EXPR, recv
 SOCKET,SCALAR,LENGTH,FLAGS, redo LABEL, redo, ref EXPR, ref, rename
 OLDNAME,NEWNAME, require VERSION, require EXPR, require, reset EXPR, reset, return
 EXPR, return, reverse LIST, rewinddir DIRHANDLE, rindex STR,SUBSTR,POSITION, rindex
 STR,SUBSTR, rmdir FILENAME, rmdir, s///, scalar EXPR, seek
 FILEHANDLE,POSITION,WHENCE, seekdir DIRHANDLE,POS, select FILEHANDLE,
 select, select RBITS,WBITS,EBITS,TIMEOUT, semctl ID,SEMNUM,CMD,ARG, semget
 KEY,NSEMS,FLAGS, semop KEY,OPSTRING, send SOCKET,MSG,FLAGS,TO, send
 SOCKET,MSG,FLAGS, setpgid PID,PGRP, setpriority WHICH,WHO,PRIORITY, setsockopt
 SOCKET,LEVEL,OPTNAME,OPTVAL, shift ARRAY, shift, shmctl ID,CMD,ARG, shmget
 KEY,SIZE,FLAGS, shmread ID,VAR,POS,SIZE, shmwrite ID,STRING,POS,SIZE, shutdown
 SOCKET,HOW, sin EXPR, sin, sleep EXPR, sleep, socket
 SOCKET,DOMAIN,TYPE,PROTOCOL, socketpair
 SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL, sort SUBNAME LIST, sort BLOCK
 LIST, sort LIST, splice ARRAY,OFFSET,LENGTH,LIST, splice ARRAY,OFFSET,LENGTH,
 splice ARRAY,OFFSET, splice ARRAY, split /PATTERN/,EXPR,LIMIT, split
 /PATTERN/,EXPR, split /PATTERN/, split, sprintf FORMAT, LIST, format parameter index,
 flags, vector flag, (minimum) width, precision, or maximum width, size, order of arguments, sqrt
 EXPR, sqrt, srand EXPR, srand, stat FILEHANDLE, stat EXPR, stat, study SCALAR, study,
 sub NAME BLOCK, sub NAME (PROTO) BLOCK, sub NAME : ATTRS BLOCK, sub NAME
 (PROTO) : ATTRS BLOCK, substr EXPR,OFFSET,LENGTH,REPLACEMENT, substr
 EXPR,OFFSET,LENGTH, substr EXPR,OFFSET, symlink OLDFILE,NEWFILE, syscall LIST,
 sysopen FILEHANDLE,FILENAME,MODE, sysopen
 FILEHANDLE,FILENAME,MODE,PERMS, sysread
 FILEHANDLE,SCALAR,LENGTH,OFFSET, sysread FILEHANDLE,SCALAR,LENGTH,
 sysseek FILEHANDLE,POSITION,WHENCE, system LIST, system PROGRAM LIST,
 syswrite FILEHANDLE,SCALAR,LENGTH,OFFSET, syswrite
 FILEHANDLE,SCALAR,LENGTH, syswrite FILEHANDLE,SCALAR, tell FILEHANDLE,
 tell, telldir DIRHANDLE, tie VARIABLE,CLASSNAME,LIST, tied VARIABLE, time, times,
 tr///, truncate FILEHANDLE,LENGTH, truncate EXPR,LENGTH, uc EXPR, uc, ucfirst EXPR,
 ucfirst, umask EXPR, umask, undef EXPR, undef, unlink LIST, unlink, unpack
 TEMPLATE,EXPR, untie VARIABLE, unshift ARRAY,LIST, use Module VERSION LIST,
 use Module VERSION, use Module LIST, use Module, use VERSION, utime LIST, values
 HASH, vec EXPR,OFFSET,BITS, wait, waitpid PID,FLAGS, wantarray, warn LIST, write
 FILEHANDLE, write EXPR, write, y///

perlreftut – Mark's very short tutorial about references

DESCRIPTION

Who Needs Complicated Data Structures?

The Solution

Syntax

Making References

Using References

An Example

Arrow Rule

Solution

The Rest

Summary

Credits

Distribution Conditions

perldsc – Perl Data Structures Cookbook

DESCRIPTION

arrays of arrays, hashes of arrays, arrays of hashes, hashes of hashes, more elaborate constructs

REFERENCES

COMMON MISTAKES

CAVEAT ON PRECEDENCE

WHY YOU SHOULD ALWAYS `use strict`

DEBUGGING

CODE EXAMPLES

ARRAYS OF ARRAYS

Declaration of an ARRAY OF ARRAYS

Generation of an ARRAY OF ARRAYS

Access and Printing of an ARRAY OF ARRAYS

HASHES OF ARRAYS

Declaration of a HASH OF ARRAYS

Generation of a HASH OF ARRAYS

Access and Printing of a HASH OF ARRAYS

ARRAYS OF HASHES

Declaration of an ARRAY OF HASHES

Generation of an ARRAY OF HASHES

Access and Printing of an ARRAY OF HASHES

HASHES OF HASHES

Declaration of a HASH OF HASHES

Generation of a HASH OF HASHES

Access and Printing of a HASH OF HASHES

MORE ELABORATE RECORDS

Declaration of MORE ELABORATE RECORDS

Declaration of a HASH OF COMPLEX RECORDS

Generation of a HASH OF COMPLEX RECORDS

Database Ties

SEE ALSO

AUTHOR

perlrequick – Perl regular expressions quick start

DESCRIPTION

The Guide

- Simple word matching
- Using character classes
- Matching this or that
- Grouping things and hierarchical matching
- Extracting matches
- Matching repetitions
- More matching
- Search and replace
- The split operator
- BUGS
- SEE ALSO
- AUTHOR AND COPYRIGHT
- Acknowledgments

perlpod – the Plain Old Documentation format

DESCRIPTION

- Ordinary Paragraph
- Verbatim Paragraph
- Command Paragraph

`=head1 Heading Text`, `=head2 Heading Text`, `=head3 Heading Text`,
`=head4 Heading Text`, `=over indentlevel`, `=item stuff...`, `=back`, `=cut`,
`=pod`, `=begin formatname`, `=end formatname`, `=for formatname text...`

Formatting Codes

`I<text>` — italic text, `B<text>` — bold text, `C<code>` — code text, `L<name>` — a
 hyperlink, `E<escape>` — a character escape, `F<filename>` — used for filenames,
`S<text>` — text contains non-breaking spaces, `X<topic name>` — an index entry, `Z<>` —
 a null (zero-effect) formatting code

- The Intent
- Embedding Pods in Perl Modules
- Hints for Writing Pod

SEE ALSO
 AUTHOR

perlpodspec – Plain Old Documentation: format specification and

notes

DESCRIPTION

Pod Definitions
 Pod Commands

`"=head1"`, `"=head2"`, `"=head3"`, `"=head4"`, `"=pod"`, `"=cut"`, `"=over"`, `"=item"`, `"=back"`, `"=begin
 formatname"`, `"=end formatname"`, `"=for formatname text..."`

Pod Formatting Codes

`I<text>` — italic text, `B<text>` — bold text, `C<code>` — code text, `F<filename>` — style for
 filenames, `X<topic name>` — an index entry, `Z<>` — a null (zero-effect) formatting code,
`L<name>` — a hyperlink, `E<escape>` — a character escape, `S<text>` — text contains
 non-breaking spaces

Notes on Implementing Pod Processors
 About `L<...>` Codes

First:, Second:, Third:, Fourth:, Fifth:, Sixth:

About `=over...=back` Regions

About Data Paragraphs and "=begin/=end" Regions

SEE ALSO

AUTHOR

perlstyle – Perl style guide

DESCRIPTION

perltrap – Perl traps for the unwary

DESCRIPTION

Awk Traps

C Traps

Sed Traps

Shell Traps

Perl Traps

Perl4 to Perl5 Traps

Discontinuance, Deprecation, and BugFix traps, Parsing Traps, Numerical Traps, General data type traps, Context Traps – scalar, list contexts, Precedence Traps, General Regular Expression Traps using `s///`, etc, Subroutine, Signal, Sorting Traps, OS Traps, DBM Traps, Unclassified Traps

Discontinuance, Deprecation, and BugFix traps

Discontinuance, Deprecation, BugFix, Discontinuance, Discontinuance, Discontinuance, BugFix, Discontinuance, Discontinuance, BugFix, Discontinuance, Deprecation, Discontinuance, Discontinuance

Parsing Traps

Parsing, Parsing, Parsing, Parsing, Parsing

Numerical Traps

Numerical, Numerical, Numerical, Bitwise string ops

General data type traps

(Arrays), (Arrays), (Hashes), (Globs), (Globs), (Scalar String), (Constants), (Scalars), (Variable Suicide)

Context Traps – scalar, list contexts

(list context), (scalar context), (scalar context), (list, builtin)

Precedence Traps

Precedence, Precedence, Precedence, Precedence, Precedence, Precedence, Precedence

General Regular Expression Traps using `s///`, etc.

Regular Expression, Regular Expression, Regular Expression, Regular Expression, Regular Expression, Regular Expression, Regular Expression, Regular Expression

Subroutine, Signal, Sorting Traps

(Signals), (Sort Subroutine), `warn()` won't let you specify a filehandle

OS Traps

(SysV), (SysV)

Interpolation Traps

Interpolation, Interpolation, Interpolation, Interpolation, Interpolation, Interpolation, Interpolation, Interpolation, Interpolation

DBM Traps

DBM, DBM

Unclassified Traps

require/do trap using returned value, split on empty string with LIMIT specified

perlrun – how to execute the Perl interpreter

SYNOPSIS

DESCRIPTION

#! and quoting on non–Unix systems

OS/2, MS–DOS, Win95/NT, Macintosh, VMS

Location of Perl

Command Switches

-0[*digits*], **-a**, **-C**, **-c**, **-d**, **-d:foo[=bar,baz]**, **-Dletters**, **-Dnumber**, **-e** *commandline*,
-Fpattern, **-h**, **-i[extension]**, **-Idirectory**, **-l[octnum]**, **-m[-]module**, **-M[-]module**,
-M[-]'module ...', **-[mM][-]module=arg[,arg]...**, **-n**, **-p**, **-P**, **-s**, **-S**, **-t**, **-T**, **-u**, **-U**, **-v**, **-V**,
-V:name, **-w**, **-W**, **-X**, **-x** *directory*

ENVIRONMENT

HOME, LOGDIR, PATH, PERL5LIB, PERL5OPT, PERLIO, :bytes, :crlf, :mmap, :perlio, :raw,
:stdio, :unix, :utf8, :win32, PERLIO_DEBUG, PERLLIB, PERL5DB, PERL5SHELL (specific to the
Win32 port), PERL_DEBUG_MSTATS, PERL_DESTRUCT_LEVEL, PERL_ENCODING,
PERL_ROOT (specific to the VMS port), SYS\$LOGIN (specific to the VMS port)

perldiag – various Perl diagnostics

DESCRIPTION

perllexwarn – Perl Lexical Warnings

DESCRIPTION

Default Warnings and Optional Warnings

What's wrong with **-w** and $\w

Controlling Warnings from the Command Line

-w, **-W**, **-X**

Backward Compatibility

Category Hierarchy

Fatal Warnings

Reporting Warnings from a Module

TODO

SEE ALSO

AUTHOR

perldebtut – Perl debugging tutorial

DESCRIPTION

use strict

Looking at data and **-w** and **v**

help

Stepping through code

Placeholder for a, w, t, T

REGULAR EXPRESSIONS

OUTPUT TIPS

CGI

GUIs

SUMMARY

SEE ALSO

AUTHOR
CONTRIBUTORS

perldebug – Perl debugging

DESCRIPTION

The Perl Debugger

Debugger Commands

h, h [command], h h, p expr, x [maxdepth] expr, V [pkg [vars]], X [vars], y [level [vars]], T, s [expr], n [expr], r, <CR, c [line|sub], l, l min+incr, l min-max, l line, l subname, -, v [line], f filename, /pattern/, ?pattern?, L [abw], S [[:!regex]], t, t expr, b, b [line] [condition], b subname [condition], b postpone subname [condition], b load filename, b compile subname, B line, B *, a [line] command, A line, A *, w expr, W expr, W *, o, o booloption ..., o anyoption? ..., o option=value ..., < ?, < [command], << command, ?, command, command, { ?, { [command], {{ command, ! number, ! -number, ! pattern, !! cmd, source file, H -number, q or ^D, R, |dbcmd, ||dbcmd, command, m expr, M, man [manpage]

Configurable Options

recallCommand, ShellBang, pager, tkRunning, signalLevel, warnLevel, dieLevel, AutoTrace, LineInfo, inhibit_exit, PrintRet, ornaments, frame, maxTraceLen, windowSize, arrayDepth, hashDepth, dumpDepth, compactDump, veryCompact, globPrint, DumpDBFiles, DumpPackages, DumpReused, quote, HighBit, undefPrint, UsageOnly, TTY, noTTY, ReadLine, NonStop

Debugger input/output

Prompt, Multiline commands, Stack backtrace, Line Listing Format, Frame listing

Debugging compile-time statements

Debugger Customization

Readline Support

Editor Support for Debugging

The Perl Profiler

Debugging regular expressions

Debugging memory usage

SEE ALSO

BUGS

perlvar – Perl predefined variables

DESCRIPTION

Predefined Names

\$ARG, \$_, \$a, \$b, \$<digits>, \$MATCH, \$&, \$PREMATCH, \$`, \$POSTMATCH, \$`, \$LAST_PAREN_MATCH, \$+, \$^N, @LAST_MATCH_END, @+, \$MULTILINE_MATCHING, \$*, HANDLE-input_line_number(EXPR), \$INPUT_LINE_NUMBER, \$NR, \$, IO::Handle-input_record_separator(EXPR), \$INPUT_RECORD_SEPARATOR, \$RS, \$/, HANDLE-autoflush(EXPR), \$OUTPUT_AUTOFLUSH, \$|, IO::Handle-output_field_separator EXPR, \$OUTPUT_FIELD_SEPARATOR, \$OFS, \$,, IO::Handle-output_record_separator EXPR, \$OUTPUT_RECORD_SEPARATOR, \$ORS, \$\\, \$LIST_SEPARATOR, \$", \$SUBSCRIPT_SEPARATOR, \$SUBSEP, \$;, \$OFMT, \$#, HANDLE-format_page_number(EXPR), \$FORMAT_PAGE_NUMBER, \$%, HANDLE-format_lines_per_page(EXPR), \$FORMAT_LINES_PER_PAGE, \$=, HANDLE-format_lines_left(EXPR), \$FORMAT_LINES_LEFT, \$-, @LAST_MATCH_START, @-, \$` is the same as substr(\$var, 0, \$-[0]), \$& is the same as substr(\$var, \$-[0], \$+[0] - \$-[0]), \$' is the same as substr(\$var, \$+[0]), \$1 is the same as substr(\$var, \$-[1], \$+[1] - \$-[1]), \$2 is the same as substr(\$var, \$-[2], \$+[2] - \$-[2]), \$3 is the same

```

as substr $var, $-[3], $+[3] - $-[3]), HANDLE-format_name(EXPR),
$FORMAT_NAME, $~, HANDLE-format_top_name(EXPR), $FORMAT_TOP_NAME, $^,
IO::Handle-format_line_break_characters EXPR, $FORMAT_LINE_BREAK_CHARACTERS,
$, IO::Handle-format_formfeed EXPR, $FORMAT_FORMFEED, $^L, $ACCUMULATOR,
$^A, $CHILD_ERROR, $?, ${^ENCODING}, $OS_ERROR, $ERRNO, $!, %!,
$EXTENDED_OS_ERROR, $^E, $EVAL_ERROR, $@, $PROCESS_ID, $PID, $$,
$REAL_USER_ID, $UID, $<, $EFFECTIVE_USER_ID, $EUID, $,
$REAL_GROUP_ID, $GID, $(, $EFFECTIVE_GROUP_ID, $EGID, $),
$PROGRAM_NAME, $0, $[, $], $COMPILING, $^C, $DEBUGGING, $^D,
$SYSTEM_FD_MAX, $^F, $^H, %^H, $INPLACE_EDIT, $^I, $^M, $OSNAME, $^O,
${^OPEN}, $PERLDB, $^P, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x100,
0x200, $LAST_REGEXP_CODE_RESULT, $^R, $EXCEPTIONS_BEING_CAUGHT, $^S,
$BASETIME, $^T, ${^TAINT}, $PERL_VERSION, $^V, $WARNING, $^W,
${^WARNING_BITS}, ${^WIDE_SYSTEM_CALLS}, $EXECUTABLE_NAME, $^X,
ARGV, $ARGV, @ARGV, ARGVOUT, @F, @INC, @_, %INC, %ENV, $ENV{expr},
%SIG, $SIG{expr}

```

Error Indicators

Technical Note on the Syntax of Variable Names

BUGS

perllo1 – Manipulating Arrays of Arrays in Perl

DESCRIPTION

Declaration and Access of Arrays of Arrays

Growing Your Own

Access and Printing

Slices

SEE ALSO

AUTHOR

perlopentut – tutorial on opening things in Perl

DESCRIPTION

Open à la shell

Simple Opens

Pipe Opens

The Minus File

Mixing Reads and Writes

Filters

Open à la C

Permissions à la mode

Obscure Open Tricks

Re-Opening Files (dups)

Dispelling the Dweomer

Paths as Opens

Single Argument Open

Playing with STDIN and STDOUT

Other I/O Issues

Opening Non-File Files

Binary Files

File Locking

IO Layers

SEE ALSO

AUTHOR and COPYRIGHT

HISTORY

perlpacktut – tutorial on pack and unpack

DESCRIPTION

The Basic Principle

Packing Text

Packing Numbers

Integers

Unpacking a Stack Frame

How to Eat an Egg on a Net

Floating point Numbers

Exotic Templates

Bit Strings

Uuencoding

Doing Sums

Unicode

Another Portable Binary Encoding

Lengths and Widths

String Lengths

Dynamic Templates

Packing and Unpacking C Structures

The Alignment Pit

Alignment, Take 2

Alignment, Take 3

Pointers for How to Use Them

Pack Recipes

Funnies Section

Authors

perlretut – Perl regular expressions tutorial

DESCRIPTION

Part 1: The basics

Simple word matching

Using character classes

Matching this or that

Grouping things and hierarchical matching

Extracting matches

Matching repetitions

Building a regexp

Using regular expressions in Perl

Part 2: Power tools

More on characters, strings, and character classes

Compiling and saving regular expressions

Embedding comments and modifiers in a regular expression

Non-capturing groupings

Looking ahead and looking behind

Using independent subexpressions to prevent backtracking

Conditional expressions

A bit of magic: executing Perl code in a regular expression

Pragmas and debugging

BUGS

SEE ALSO

AUTHOR AND COPYRIGHT

Acknowledgments

perlre – Perl regular expressions

DESCRIPTION

i, m, s, x

Regular Expressions

[1], [2], [3], cntrl, graph, print, punct, xdigit

Extended Patterns

```
(?#text), (?imsx-imsx), (?:pattern), (?imsx-imsx:pattern),
(?=pattern), (?!pattern), (?<=pattern), (?<!pattern), (?{ code }),
(??{ code }), < (?pattern), (? (condition)yes-pattern|no-pattern),
(? (condition)yes-pattern)
```

Backtracking

Version 8 Regular Expressions

Warning on \1 vs \$1

Repeated patterns matching zero-length substring

Combining pieces together

```
ST, S|T, S{REPEAT_COUNT}, S{min,max}, S{min,max}?, S?, S*, S+, S??, S*?, S+?,
< (?S), (?=S), (?<=S), (?!S), (?<!S), (??{ EXPR }),
(? (condition)yes-pattern|no-pattern)
```

Creating custom RE engines

BUGS

SEE ALSO

perlref – Perl references and nested data structures

NOTE

DESCRIPTION

Making References

Using References

Symbolic references

Not-so-symbolic references

Pseudo-hashes: Using an array as a hash

Function Templates

WARNING

SEE ALSO

perlform – Perl formats

DESCRIPTION

Format Variables

NOTES

Footers

Accessing Formatting Internals

WARNINGS

perlboot – Beginner's Object-Oriented Tutorial

DESCRIPTION

If we could talk to the animals...

Introducing the method invocation arrow

Invoking a barnyard

The extra parameter of method invocation

- Calling a second method to simplify things
- Inheriting the windpipes
- A few notes about @ISA
- Overriding the methods
- Starting the search from a different place
- The SUPER way of doing things
- Where we're at so far...
- A horse is a horse, of course of course — or is it?
- Invoking an instance method
- Accessing the instance data
- How to build a horse
- Inheriting the constructor
- Making a method work with either classes or instances
- Adding parameters to a method
- More interesting instances
- A horse of a different color
- Summary

SEE ALSO

COPYRIGHT

perltoot – Tom's object-oriented tutorial for perl

DESCRIPTION

- Creating a Class
 - Object Representation
 - Class Interface
 - Constructors and Instance Methods
 - Planning for the Future: Better Constructors
 - Destructors
 - Other Object Methods
- Class Data
 - Accessing Class Data
 - Debugging Methods
 - Class Destructors
 - Documenting the Interface
- Aggregation
- Inheritance
 - Overridden Methods
 - Multiple Inheritance
 - UNIVERSAL: The Root of All Objects
- Alternate Object Representations
 - Arrays as Objects
 - Closures as Objects
- AUTOLOAD: Proxy Methods
 - Autoloaded Data Methods
 - Inherited Autoloaded Data Methods
- Metaclassical Tools
 - Class::Struct
 - Data Members as Variables

NOTES

- Object Terminology

SEE ALSO

AUTHOR AND COPYRIGHT

COPYRIGHT

Acknowledgments

perltoc – Tom’s OO Tutorial for Class Data in Perl

DESCRIPTION

Class Data in a Can

Class Data as Package Variables

Putting All Your Eggs in One Basket

Inheritance Concerns

The Eponymous Meta–Object

Indirect References to Class Data

Monadic Classes

Translucent Attributes

Class Data as Lexical Variables

Privacy and Responsibility

File–Scoped Lexicals

More Inheritance Concerns

Locking the Door and Throwing Away the Key

Translucency Revisited

NOTES

SEE ALSO

AUTHOR AND COPYRIGHT

ACKNOWLEDGEMENTS

HISTORY

perlobj – Perl objects

DESCRIPTION

An Object is Simply a Reference

A Class is Simply a Package

A Method is Simply a Subroutine

Method Invocation

Indirect Object Syntax

Default UNIVERSAL methods

isa(CLASS), can(METHOD), VERSION([NEED])

Destructors

Summary

Two–Phased Garbage Collection

SEE ALSO

perlbot – Bag’o Object Tricks (the BOT)

DESCRIPTION

OO SCALING TIPS

INSTANCE VARIABLES

SCALAR INSTANCE VARIABLES

INSTANCE VARIABLE INHERITANCE

OBJECT RELATIONSHIPS

OVERRIDING SUPERCLASS METHODS

USING RELATIONSHIP WITH SDBM

THINKING OF CODE REUSE

CLASS CONTEXT AND THE OBJECT

INHERITING A CONSTRUCTOR

DELEGATION

perltie – how to hide an object class in a simple variable

SYNOPSIS

DESCRIPTION

Tying Scalars

TIESCALAR classname, LIST, FETCH this, STORE this, value, UNTIE this, DESTROY this

Tying Arrays

TIEARRAY classname, LIST, FETCH this, index, STORE this, index, value, FETCHSIZE this, STORESIZE this, count, EXTEND this, count, EXISTS this, key, DELETE this, key, CLEAR this, PUSH this, LIST, POP this, SHIFT this, UNSHIFT this, LIST, SPLICE this, offset, length, LIST, UNTIE this, DESTROY this

Tying Hashes

USER, HOME, CLOBBER, LIST, TIEHASH classname, LIST, FETCH this, key, STORE this, key, value, DELETE this, key, CLEAR this, EXISTS this, key, FIRSTKEY this, NEXTKEY this, lastkey, UNTIE this, DESTROY this

Tying FileHandles

TIEHANDLE classname, LIST, WRITE this, LIST, PRINT this, LIST, PRINTF this, LIST, READ this, LIST, READLINE this, GETC this, CLOSE this, UNTIE this, DESTROY this

UNTIE this

The `untie` Gotcha

SEE ALSO

BUGS

AUTHOR

perlipc – Perl interprocess communication (signals, fifos, pipes,

safe subprocesses, sockets, and semaphores)

DESCRIPTION

Signals

Handling the SIGHUP Signal in Daemons

Named Pipes

Deferred Signals

Long running opcodes, Interrupting IO, Signals as "faults", Signals triggered by operating system state

Using `open()` for IPC

Filehandles

Background Processes

Complete Dissociation of Child from Parent

Safe Pipe Opens

Bidirectional Communication with Another Process

Bidirectional Communication with Yourself

Sockets: Client/Server Communication

Internet Line Terminators

Internet TCP Clients and Servers

Unix-Domain TCP Clients and Servers

TCP Clients with `IO::Socket`

A Simple Client

`Proto`, `PeerAddr`, `PeerPort`

A Webget Client

- Interactive Client with IO::Socket
- TCP Servers with IO::Socket
 - Proto, LocalPort, Listen, Reuse
- UDP: Message Passing
- SysV IPC
- NOTES
- BUGS
- AUTHOR
- SEE ALSO

perlfork – Perl's `fork()` emulation

SYNOPSIS

DESCRIPTION

Behavior of other Perl features in forked pseudo-processes

`$$` or `$PROCESS_ID`, `%ENV`, `chdir()` and all other builtins that accept filenames, `wait()` and `waitpid()`, `kill()`, `exec()`, `exit()`, Open handles to files, directories and network sockets

Resource limits

Killing the parent process

Lifetime of the parent process and pseudo-processes

CAVEATS AND LIMITATIONS

BEGIN blocks, Open filehandles, Forking pipe `open()` not yet implemented, Global state maintained by XSUBs, Interpreter embedded in larger application, Thread-safety of extensions

BUGS

AUTHOR

SEE ALSO

perlnumber – semantics of numbers and numeric operations in Perl

SYNOPSIS

DESCRIPTION

Storing numbers

Numeric operators and numeric conversions

Flavors of Perl numeric operations

Arithmetic operators, `++`, Arithmetic operators during use `integer`, Other mathematical operators, Bitwise operators, Bitwise operators during use `integer`, Operators which expect an integer, Operators which expect a string

AUTHOR

SEE ALSO

perlthrtut – tutorial on threads in Perl

DESCRIPTION

Status

What Is A Thread Anyway?

Threaded Program Models

- Boss/Worker

- Work Crew

- Pipeline

Native threads

What kind of threads are Perl threads?

Thread-Safe Modules

- Thread Basics
 - Basic Thread Support
 - A Note about the Examples
 - Creating Threads
 - Giving up control
 - Waiting For A Thread To Exit
 - Ignoring A Thread
- Threads And Data
 - Shared And Unshared Data
 - Thread Pitfalls: Races
- Synchronization and control
 - Controlling access: `lock()`
 - A Thread Pitfall: Deadlocks
 - Queues: Passing Data Around
 - Semaphores: Synchronizing Data Access
 - Basic semaphores
 - Advanced Semaphores
 - `cond_wait()` and `cond_signal()`
- General Thread Utility Routines
 - What Thread Am I In?
 - Thread IDs
 - Are These Threads The Same?
 - What Threads Are Running?
- A Complete Example
- Performance considerations
- Process-scope Changes
- Thread-Safety of System Libraries
- Conclusion
- Bibliography
 - Introductory Texts
 - OS-Related References
 - Other References
- Acknowledgements
- AUTHOR
- Copyrights

perlothrtut – old tutorial on threads in Perl

- DESCRIPTION
- What Is A Thread Anyway?
- Threaded Program Models
 - Boss/Worker
 - Work Crew
 - Pipeline
- Native threads
- What kind of threads are perl threads?
- Threadsafe Modules
- Thread Basics
 - Basic Thread Support
 - Creating Threads
 - Giving up control
 - Waiting For A Thread To Exit
 - Errors In Threads
 - Ignoring A Thread

- Threads And Data
 - Shared And Unshared Data
 - Thread Pitfall: Races
 - Controlling access: `lock()`
 - Thread Pitfall: Deadlocks
 - Queues: Passing Data Around
- Threads And Code
 - Semaphores: Synchronizing Data Access
 - Basic semaphores, Advanced Semaphores
 - Attributes: Restricting Access To Subroutines
 - Subroutine Locks
 - Methods
 - Locking A Subroutine
- General Thread Utility Routines
 - What Thread Am I In?
 - Thread IDs
 - Are These Threads The Same?
 - What Threads Are Running?
- A Complete Example
- Conclusion
- Bibliography
 - Introductory Texts
 - OS-Related References
 - Other References
- Acknowledgements
- AUTHOR
- Copyrights

perlport – Writing portable Perl

DESCRIPTION

Not all Perl programs have to be portable, Nearly all of Perl already *is* portable

ISSUES

- Newlines
- Numbers endianness and Width
- Files and Filesystems
- System Interaction
- Command names versus file pathnames
- Interprocess Communication (IPC)
- External Subroutines (XS)
- Standard Modules
- Time and Date
- Character sets and character encoding
- Internationalisation
- System Resources
- Security
- Style

CPAN Testers

Mailing list: cpan-testers@perl.org, Testing results: <http://testers.cpan.org/>

PLATFORMS

- Unix
- DOS and Derivatives

Mac OS

VMS

VOS

EBCDIC Platforms

Acorn RISC OS

Other perls

FUNCTION IMPLEMENTATIONS

Alphabetical Listing of Perl Functions

-X FILEHANDLE, -X EXPR, -X, alarm SECONDS, alarm, binmode FILEHANDLE, chmod
 LIST, chown LIST, chroot FILENAME, chroot, crypt PLAINTEXT,SALT, dbmclose HASH,
 dbmopen HASH,DBNAME,MODE, dump LABEL, exec LIST, exit EXPR, exit, fcntl
 FILEHANDLE,FUNCTION,SCALAR, flock FILEHANDLE,OPERATION, fork, getlogin,
 getpgrp PID, getppid, getpriority WHICH,WHO, getpwnam NAME, getgrnam NAME,
 getnetbyname NAME, getpwuid UID, getgrgid GID, getnetbyaddr ADDR,ADDRTYPE,
 getprotobynumber NUMBER, getservbyport PORT,PROTO, getpwent, getgrent, gethostent,
 getnetent, getprotoent, getservent, sethostent STAYOPEN, setnetent STAYOPEN, setprotoent
 STAYOPEN, setservent STAYOPEN, endpwent, endgrent, endhostent, endnetent, endprotoent,
 endservent, getsockopt SOCKET,LEVEL,OPTNAME, glob EXPR, glob, ioctl
 FILEHANDLE,FUNCTION,SCALAR, kill SIGNAL, LIST, link OLDFILE,NEWFILE, lstat
 FILEHANDLE, lstat EXPR, lstat, msgctl ID,CMD,ARG, msgget KEY,FLAGS, msgsnd
 ID,MSG,FLAGS, msgrcv ID,VAR,SIZE,TYPE,FLAGS, open FILEHANDLE,EXPR, open
 FILEHANDLE, pipe READHANDLE,WRITEHANDLE, readlink EXPR, readlink, select
 RBITS,WBITS,EBITS,TIMEOUT, semctl ID,SEMNUM,CMD,ARG, semget
 KEY,NSEMS,FLAGS, semop KEY,OPSTRING, setgrent, setpgrp PID,PGRP, setpriority
 WHICH,WHO,PRIORITY, setpwent, setsockopt SOCKET,LEVEL,OPTNAME,OPTVAL,
 shmctl ID,CMD,ARG, shmget KEY,SIZE,FLAGS, shmread ID,VAR,POS,SIZE, shmwrite
 ID,STRING,POS,SIZE, socketatmark SOCKET, socketpair
 SOCKET1,SOCKET2,DOMAIN,TYPE,PROTOCOL, stat FILEHANDLE, stat EXPR, stat,
 symlink OLDFILE,NEWFILE, syscall LIST, sysopen
 FILEHANDLE,FILENAME,MODE,PERMS, system LIST, times, truncate
 FILEHANDLE,LENGTH, truncate EXPR,LENGTH, umask EXPR, umask, utime LIST, wait,
 waitpid PID,FLAGS

CHANGES

v1.48, 02 February 2001, v1.47, 22 March 2000, v1.46, 12 February 2000, v1.45, 20 December 1999,
 v1.44, 19 July 1999, v1.43, 24 May 1999, v1.42, 22 May 1999, v1.41, 19 May 1999, v1.40, 11 April
 1999, v1.39, 11 February 1999, v1.38, 31 December 1998, v1.37, 19 December 1998, v1.36, 9
 September 1998, v1.35, 13 August 1998, v1.33, 06 August 1998, v1.32, 05 August 1998, v1.30, 03
 August 1998, v1.23, 10 July 1998

Supported Platforms

SEE ALSO

AUTHORS / CONTRIBUTORS

perllocale – Perl locale handling (internationalization and

localization)

DESCRIPTION

PREPARING TO USE LOCALES

USING LOCALES

The use locale pragma

The setlocale function

Finding locales

LOCALE PROBLEMS

- Temporarily fixing locale problems
- Permanently fixing locale problems
- Permanently fixing your system's locale configuration
- Fixing system locale configuration
- The localeconv function
- I18N::Langinfo
- LOCALE CATEGORIES
 - Category LC_COLLATE: Collation
 - Category LC_CTYPE: Character Types
 - Category LC_NUMERIC: Numeric Formatting
 - Category LC_MONETARY: Formatting of monetary amounts
 - LC_TIME
 - Other categories
- SECURITY
- ENVIRONMENT
 - PERL_BADLANG, LC_ALL, LANGUAGE, LC_CTYPE, LC_COLLATE, LC_MONETARY, LC_NUMERIC, LC_TIME, LANG
- NOTES
 - Backward compatibility
 - I18N:Collate obsolete
 - Sort speed and memory use impacts
 - write() and LC_NUMERIC
 - Freely available locale definitions
 - I18n and I10n
 - An imperfect standard
- Unicode and UTF-8
- BUGS
 - Broken systems
- SEE ALSO
- HISTORY

perluniintro – Perl Unicode introduction

- DESCRIPTION
 - Unicode
 - Perl's Unicode Support
 - Perl's Unicode Model
 - Unicode and EBCDIC
 - Creating Unicode
 - Handling Unicode
 - Legacy Encodings
 - Unicode I/O
 - Displaying Unicode As Text
 - Special Cases
 - Advanced Topics
 - Miscellaneous
 - Questions With Answers
 - Hexadecimal Notation
 - Further Resources
- UNICODE IN OLDER PERLS
- SEE ALSO
- ACKNOWLEDGMENTS
- AUTHOR, COPYRIGHT, AND LICENSE

perlunicode – Unicode support in Perl

DESCRIPTION

Important Caveats

Input and Output Layers, Regular Expressions, use utf8 still needed to enable UTF-8/UTF-EBCDIC in scripts

Byte and Character Semantics

Effects of Character Semantics

Scripts

Blocks

User-Defined Character Properties

Character Encodings for Input and Output

Unicode Regular Expression Support Level

Unicode Encodings

Security Implications of Unicode

Unicode in Perl on EBCDIC

Locales

Using Unicode in XS

BUGS

Interaction with Locales

Interaction with Extensions

Speed

SEE ALSO

perlebcdic – Considerations for running Perl on EBCDIC platforms

DESCRIPTION

COMMON CHARACTER CODE SETS

ASCII

ISO 8859

Latin 1 (ISO 8859-1)

EBCDIC

13 variant characters

0037

1047

POSIX-BC

Unicode code points versus EBCDIC code points

Remaining Perl Unicode problems in EBCDIC

Unicode and UTF

Using Encode

SINGLE OCTET TABLES

recipe 0, recipe 1, recipe 2, recipe 3, recipe 4, recipe 5, recipe 6

IDENTIFYING CHARACTER CODE SETS

CONVERSIONS

tr///

iconv

C RTL

OPERATOR DIFFERENCES

FUNCTION DIFFERENCES

chr(), ord(), pack(), print(), printf(), sort(), sprintf(), unpack()

REGULAR EXPRESSION DIFFERENCES

SOCKETS

SORTING

Ignore ASCII vs. EBCDIC sort differences.
MONO CASE then sort data.
Convert, sort data, then re convert.
Perform sorting on one type of machine only.

TRANSFORMATION FORMATS

URL decoding and encoding
uu encoding and decoding
Quoted-Printable encoding and decoding
Caesarian ciphers

Hashing order and checksums

I18N AND L10N

MULTI OCTET CHARACTER SETS**OS ISSUES**

OS/400

IFS access

OS/390, z/OS

chcp, dataset access, OS/390, z/OS iconv, locales

VM/ESA?

POSIX-BC?

BUGS**SEE ALSO****REFERENCES****HISTORY****AUTHOR****perlsec – Perl security****DESCRIPTION**

Laundering and Detecting Tainted Data
Switches On the "#!" Line
Cleaning Up Your Path
Security Bugs
Protecting Your Programs
Unicode

SEE ALSO**perlmod – Perl modules (packages and symbol tables)****DESCRIPTION**

Packages
Symbol Tables
Package Constructors and Destructors
Perl Classes
Perl Modules
Making your module threadsafe

SEE ALSO**perlmodinstall – Installing CPAN Modules****DESCRIPTION****PREAMBLE**

DECOMPRESS the file, **UNPACK** the file into a directory, **BUILD** the module (sometimes unnecessary), **INSTALL** the module

PORTABILITY
HEY
AUTHOR
COPYRIGHT

perlmodlib – constructing new Perl modules and finding existing ones

DESCRIPTION

THE PERL MODULE LIBRARY

Pragmatic Modules

attributes, attrs, autouse, base, bigint, bignum, bigrat, blib, bytes, charnames, constant, diagnostics, encoding, fields, filetest, if, integer, less, locale, open, ops, overload, re, sigtrap, sort, strict, subs, threads, utf8, vars, vmsish, warnings, warnings::register

Standard Modules

AnyDBM_File, Attribute::Handlers, AutoLoader, AutoSplit, B, B::Asmdata, B::Assembler, B::Block, B::Bytecode, B::C, B::CC, B::Concise, B::Debug, B::Deparse, B::Disassembler, B::Lint, B::Showlex, B::Stackobj, B::Stash, B::Terse, B::Xref, Benchmark, ByteLoader, CGI, CGI::Apache, CGI::Carp, CGI::Cookie, CGI::Fast, CGI::Pretty, CGI::Push, CGI::Switch, CGI::Util, CPAN, CPAN::FirstTime, CPAN::Nox, Carp, Carp::Heavy, Class::ISA, Class::Struct, Config, Cwd, DB, DB_File, Devel::SelfStubber, Digest, DirHandle, Dumpvalue, Encode, English, Env, Errno, Exporter, Exporter::Heavy, ExtUtils::Command, ExtUtils::Command::MM, ExtUtils::Constant, ExtUtils::Embed, ExtUtils::Install, ExtUtils::Installed, ExtUtils::Liblist, ExtUtils::MM, ExtUtils::MM_Any, ExtUtils::MM_BeOS, ExtUtils::MM_Cygwin, ExtUtils::MM_DOS, ExtUtils::MM_MacOS, ExtUtils::MM_NW5, ExtUtils::MM_OS2, ExtUtils::MM_UWIN, ExtUtils::MM_Unix, ExtUtils::MM_VMS, ExtUtils::MM_Win32, ExtUtils::MM_Win95, ExtUtils::MY, ExtUtils::MakeMaker, ExtUtils::Manifest, ExtUtils::Mkbootstrap, ExtUtils::Mksymlists, ExtUtils::Packlist, ExtUtils::testlib, Fatal, Fcntl, File::Basename, File::CheckTree, File::Compare, File::Copy, File::DosGlob, File::Find, File::Path, File::Spec, File::Spec::Cygwin, File::Spec::Epoc, File::Spec::Functions, File::Spec::Mac, File::Spec::OS2, File::Spec::Unix, File::Spec::VMS, File::Spec::Win32, File::Temp, File::stat, FileCache, FileHandle, Filter::Simple, FindBin, Getopt::Long, Getopt::Std, Hash::Util, I18N::Collate, I18N::LangTags, I18N::LangTags::List, IO, IPC::Open2, IPC::Open3, Locale::Constants, Locale::Country, Locale::Currency, Locale::Language, Locale::Maketext, Locale::Maketext::TPJ13, Locale::Script, Math::BigFloat, Math::BigInt, Math::BigInt::Calc, Math::BigRat, Math::Complex, Math::Trig, Memoize, Memoize::AnyDBM_File, Memoize::Expire, Memoize::ExpireFile, Memoize::ExpireTest, Memoize::NDBM_File, Memoize::SDBM_File, Memoize::Storable, NDBM_File, NEXT, Net::Cmd, Net::Config, Net::Domain, Net::FTP, Net::NNTP, Net::Netrc, Net::POP3, Net::Ping, Net::SMTP, Net::Time, Net::hostent, Net::libnetFAQ, Net::netent, Net::protoent, Net::servent, O, ODBM_File, Opcode, POSIX, PerlIO, PerlIO::via::QuotedPrint, Pod::Checker, Pod::Find, Pod::Functions, Pod::Html, Pod::InputObjects, Pod::LaTeX, Pod::Man, Pod::ParseLink, Pod::ParseUtils, Pod::Parser, Pod::Plainer, Pod::Select, Pod::Text, Pod::Text::Color, Pod::Text::Overstrike, Pod::Text::Termcap, Pod::Usage, SDBM_File, Safe, Search::Dict, SelectSaver, SelfLoader, Shell, Socket, Storable, Switch, Symbol, Term::ANSIColor, Term::Cap, Term::Complete, Term::ReadLine, Test, Test::Builder, Test::Harness, Test::Harness::Assert, Test::Harness::Iterator, Test::Harness::Straps, Test::More, Test::Simple, Test::Tutorial, Text::Abbrev, Text::Balanced, Text::ParseWords, Text::Soundex, Text::Tabs, Text::Wrap, Thread, Thread::Queue, Thread::Semaphore, Tie::Array, Tie::File, Tie::Handle, Tie::Hash, Tie::Memoize, Tie::RefHash, Tie::Scalar, Tie::SubstrHash, Time::Local, Time::gmtime, Time::localtime, Time::tm, UNIVERSAL, Unicode::Collate, Unicode::UCD, User::grent, User::pwent, Win32

Extension Modules

CPAN

- Africa
 - South Africa
- Asia
 - China, India, Indonesia, Israel, Japan, Korea, Philippines, Russian Federation, Saudi Arabia, Singapore, South Korea, Taiwan, Thailand
- Central America
 - Costa Rica
- Europe
 - Austria, Belgium, Bulgaria, Croatia, Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Netherlands, Norway, Poland, Portugal, Romania, Russia, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey, Ukraine, United Kingdom
- North America
 - Alberta, Manitoba, Nova Scotia, Ontario, Quebec, Mexico
- United States
 - Alabama, California, Colorado, Delaware, District of Columbia, Florida, Illinois, Indiana, Kentucky, Massachusetts, Michigan, New Jersey, New York, North Carolina, Ohio, Oklahoma, Oregon, Pennsylvania, Tennessee, Texas, Utah, Virginia, Washington, Wisconsin
- Oceania
 - Australia, New Zealand
- South America
 - Argentina, Brazil, Chile

RSYNC Mirrors

Modules: Creation, Use, and Abuse

- Guidelines for Module Creation
- Guidelines for Converting Perl 4 Library Scripts into Modules
- Guidelines for Reusing Application Code

NOTE

perlmodstyle – Perl module style guide

INTRODUCTION

QUICK CHECKLIST

- Before you start
- The API
- Stability
- Documentation
- Release considerations

BEFORE YOU START WRITING A MODULE

- Has it been done before?
- Do one thing and do it well
- What's in a name?

DESIGNING AND WRITING YOUR MODULE

- To OO or not to OO?
- Designing your API
 - Write simple routines to do simple things, Separate functionality from output, Provide sensible shortcuts and defaults, Naming conventions, Parameter passing

- Strictness and warnings
- Backwards compatibility
- Error handling and messages
- DOCUMENTING YOUR MODULE
 - POD
 - README, INSTALL, release notes, changelogs
- RELEASE CONSIDERATIONS
 - Version numbering
 - Pre-requisites
 - Testing
 - Packaging
 - Licensing
- COMMON PITFALLS
 - Reinventing the wheel
 - Trying to do too much
 - Inappropriate documentation
- SEE ALSO

[perlstyle](#), [perlnewmod](#), [perlpod](#), [podchecker](#), Testing tools, <http://pause.perl.org/>, Any good book on software engineering

AUTHOR

perlnewmod – preparing a new module for distribution

DESCRIPTION

- Warning
- What should I make into a module?
- Step-by-step: Preparing the ground
 - Look around, Check it's new, Discuss the need, Choose a name, Check again
- Step-by-step: Making the module
 - Start with ***h2xs***, Use *strict/strict* and *warnings/warnings*, Use *Carp/Carp*, Use *Exporter/Exporter* – wisely!, Use *plain old documentation/perlpod*, Write tests, Write the README
- Step-by-step: Distributing your module
 - Get a CPAN user ID, perl Makefile.PL; make test; make dist, Upload the tarball, Announce to the modules list, Announce to clpa, Fix bugs!

AUTHOR

SEE ALSO

perlfaq1 – General Questions About Perl (\$Revision: 1.8 \$, \$Date:

2002/04/07 18:46:13 \$)

DESCRIPTION

- What is Perl?
- Who supports Perl? Who develops it? Why is it free?
- Which version of Perl should I use?
- What are perl4 and perl5?
- What is perl6?
- How stable is Perl?
- Is Perl difficult to learn?
- How does Perl compare with other languages like Java, Python, REXX, Scheme, or Tcl?

Can I do [task] in Perl?
When shouldn't I program in Perl?
What's the difference between "perl" and "Perl"?
Is it a Perl program or a Perl script?
What is a JAPH?
Where can I get a list of Larry Wall witticisms?
How can I convince my sysadmin/supervisor/employees to use version
5/5.6.1/Perl instead of some other language?

AUTHOR AND COPYRIGHT

**perlfaq2 – Obtaining and Learning about Perl (\$Revision: 1.13 \$,
\$Date: 2002/04/26 16:56:35 \$)**

DESCRIPTION

What machines support Perl? Where do I get it?
How can I get a binary version of Perl?
I don't have a C compiler on my system. How can I compile perl?
I copied the Perl binary from one machine to another, but scripts
don't work.

I grabbed the sources and tried to compile but gdbm/dynamic
loading/malloc/linking/... failed. How do I make it work?

What modules and extensions are available for Perl? What is CPAN?
What does CPAN/src/... mean?

Is there an ISO or ANSI certified version of Perl?
Where can I get information on Perl?
What are the Perl newsgroups on Usenet? Where do I post questions?
Where should I post source code?
Perl Books
References, Tutorials, Task-Oriented, Special Topics

Perl in Magazines
Perl on the Net: FTP and WWW Access
What mailing lists are there for Perl?
Archives of comp.lang.perl.misc
Where can I buy a commercial version of Perl?
Where do I send bug reports?
What is perl.com? Perl Mongers? pm.org? perl.org? cpan.org?

AUTHOR AND COPYRIGHT

**perlfaq3 – Programming Tools (\$Revision: 1.22 \$, \$Date: 2002/05/06
13:11:13 \$)**

DESCRIPTION

How do I do (anything)?
How can I use Perl interactively?
Is there a Perl shell?
How do I debug my Perl programs?
How do I profile my Perl programs?
How do I cross-reference my Perl programs?
Is there a pretty-printer (formatter) for Perl?
Is there a ctags for Perl?

Is there an IDE or Windows Perl Editor?

Komodo, The Object System, Open Perl IDE, PerlBuilder, visiPerl+, OptiPerl, CodeMagicCD, GNU Emacs, MicroEMACS, XEmacs, Elvis, Vile, Vim, Codewright, MultiEdit, SlickEdit, Bash, Ksh, Tcsh, Zsh, BBEEdit and BBEEdit Lite, Alpha

Where can I get Perl macros for vi?

Where can I get perl-mode for emacs?

How can I use curses with Perl?

How can I use X or Tk with Perl?

How can I generate simple menus without using CGI or Tk?

How can I make my Perl program run faster?

How can I make my Perl program take less memory?

Don't slurp!, Use map and grep selectively, Avoid unnecessary quotes and stringification, Pass by reference, Tie large variables to disk

Is it unsafe to return a pointer to local data?

How can I free an array or hash so my program shrinks?

How can I make my CGI script more efficient?

How can I hide the source for my Perl program?

How can I compile my Perl program into byte code or C?

How can I compile Perl into Java?

How can I get `#!perl` to work on [MS-DOS,NT,...]?

Can I write useful Perl programs on the command line?

Why don't Perl one-liners work on my DOS/Mac/VMS system?

Where can I learn about CGI or Web programming in Perl?

Where can I learn about object-oriented Perl programming?

Where can I learn about linking C with Perl? [h2xs, xsubpp]

I've read perlembed, perlguits, etc., but I can't embed perl in my C program; what am I doing wrong?

When I tried to run my script, I got this message. What does it mean?

What's MakeMaker?

AUTHOR AND COPYRIGHT

perlfaq4 – Data Manipulation (\$Revision: 1.25 \$, \$Date: 2002/05/30

07:04:25 \$)

DESCRIPTION

Data: Numbers

Why am I getting long decimals (eg, 19.9499999999999) instead of the numbers I should be getting (eg, 19.95)?

Why isn't my octal data interpreted correctly?

Does Perl have a `round()` function? What about `ceil()` and `floor()`?

Trig functions?

How do I convert between numeric representations?

How do I convert hexadecimal into decimal, How do I convert from decimal to hexadecimal, How do I convert from octal to decimal, How do I convert from decimal to octal, How do I convert from binary to decimal, How do I convert from decimal to binary

Why doesn't `&` work the way I want it to?

How do I multiply matrices?

How do I perform an operation on a series of integers?

How can I output Roman numerals?

- Why aren't my random numbers random?
- How do I get a random number between X and Y?
- Data: Dates
 - How do I find the week-of-the-year/day-of-the-year?
 - How do I find the current century or millennium?
 - How can I compare two dates and find the difference?
 - How can I take a string and turn it into epoch seconds?
 - How can I find the Julian Day?
 - How do I find yesterday's date?
 - Does Perl have a Year 2000 problem? Is Perl Y2K compliant?
- Data: Strings
 - How do I validate input?
 - How do I unescape a string?
 - How do I remove consecutive pairs of characters?
 - How do I expand function calls in a string?
 - How do I find matching/nesting anything?
 - How do I reverse a string?
 - How do I expand tabs in a string?
 - How do I reformat a paragraph?
 - How can I access/change the first N letters of a string?
 - How do I change the Nth occurrence of something?
 - How can I count the number of occurrences of a substring within a string?
 - How do I capitalize all the words on one line?
 - How can I split a [character] delimited string except when inside [character]? (Comma-separated files)
 - How do I strip blank space from the beginning/end of a string?
 - How do I pad a string with blanks or pad a number with zeroes?
 - How do I extract selected columns from a string?
 - How do I find the soundex value of a string?
 - How can I expand variables in text strings?
 - What's wrong with always quoting "\$vars"?
 - Why don't my <<HERE documents work?
 1. There must be no space after the << part,
 2. There (probably) should be a semicolon at the end,
 3. You can't (easily) have any space in front of the tag
- Data: Arrays
 - What is the difference between a list and an array?
 - What is the difference between `$array[1]` and `@array[1]`?
 - How can I remove duplicate elements from a list or array?
 - a), b), c), d), e)
 - How can I tell whether a certain element is contained in a list or array?
 - How do I compute the difference of two arrays? How do I compute the intersection of two arrays?
 - How do I test whether two arrays or hashes are equal?
 - How do I find the first array element for which a condition is true?
 - How do I handle linked lists?
 - How do I handle circular lists?
 - How do I shuffle an array randomly?

How do I process/modify each element of an array?
How do I select a random element from an array?
How do I permute N elements of a list?
How do I sort an array by (anything)?
How do I manipulate arrays of bits?
Why does `defined()` return true on empty arrays and hashes?
Data: Hashes (Associative Arrays)
How do I process an entire hash?
What happens if I add or remove keys from a hash while iterating over it?

How do I look up a hash element by value?
How can I know how many entries are in a hash?
How do I sort a hash (optionally by value instead of key)?
How can I always keep my hash sorted?
What's the difference between "delete" and "undef" with hashes?
Why don't my tied hashes make the defined/exists distinction?
How do I reset an `each()` operation part-way through?
How can I get the unique keys from two hashes?
How can I store a multidimensional array in a DBM file?
How can I make my hash remember the order I put elements into it?
Why does passing a subroutine an undefined element in a hash create it?

How can I make the Perl equivalent of a C structure/C++ class/hash or array of hashes or arrays?

How can I use a reference as a hash key?
Data: Misc
How do I handle binary data correctly?
How do I determine whether a scalar is a number/whole/integer/float?
How do I keep persistent data across program calls?
How do I print out or copy a recursive data structure?
How do I define methods for every class/object?
How do I verify a credit card checksum?
How do I pack arrays of doubles or floats for XS code?
AUTHOR AND COPYRIGHT

perlfac5 – Files and Formats (\$Revision: 1.18 \$, \$Date: 2002/05/30

07:04:25 \$)

DESCRIPTION

How do I flush/unbuffer an output filehandle? Why must I do this?
How do I change one line in a file/delete a line in a file/insert a line in the middle of a file/append to the beginning of a file?

How do I count the number of lines in a file?
How do I make a temporary file name?
How can I manipulate fixed-record-length files?
How can I make a filehandle local to a subroutine? How do I pass filehandles between subroutines? How do I make an array of filehandles?

How can I use a filehandle indirectly?
How can I set up a footer format to be used with `write()`?
How can I `write()` into a string?

How can I output my numbers with commas added?
 How can I translate tildes (~) in a filename?
 How come when I open a file read-write it wipes it out?
 Why do I sometimes get an "Argument list too long" when I use <*?
 Is there a leak/bug in `glob()`?
 How can I open a file with a leading "" or trailing blanks?
 How can I reliably rename a file?
 How can I lock a file?
 Why can't I just open(FH, "file.lock")?
 I still don't get locking. I just want to increment the number in
 the file. How can I do this?

All I want to do is append a small amount of text to the end of a
 file. Do I still have to use locking?

How do I randomly update a binary file?
 How do I get a file's timestamp in perl?
 How do I set a file's timestamp in perl?
 How do I print to more than one file at once?
 How can I read in an entire file all at once?
 How can I read in a file by paragraphs?
 How can I read a single character from a file? From the keyboard?
 How can I tell whether there's a character waiting on a filehandle?
 How do I do a `tail -f` in perl?
 How do I `dup()` a filehandle in Perl?
 How do I close a file descriptor by number?
 Why can't I use "C:\temp\foo" in DOS paths? Why doesn't
 'C:\temp\foo.exe' work?
 Why doesn't `glob("*.*)` get all the files?
 Why does Perl let me delete read-only files? Why does `-i` clobber
 protected files? Isn't this a bug in Perl?
 How do I select a random line from a file?
 Why do I get weird spaces when I print an array of lines?

AUTHOR AND COPYRIGHT

perlfac6 – Regular Expressions (\$Revision: 1.12 \$, \$Date: 2002/06/01

22:31:09 \$)

DESCRIPTION

How can I hope to use regular expressions without creating illegible
 and unmaintainable code?

Comments Outside the Regex, Comments Inside the Regex, Different Delimiters

I'm having trouble matching over more than one line. What's wrong?
 How can I pull out lines between two patterns that are themselves on
 different lines?

I put a regular expression into `/` but it didn't work. What's wrong?
 How do I substitute case insensitively on the LHS while preserving
 case on the RHS?

How can I make `\w` match national character sets?
 How can I match a locale-smart version of `/ [a-zA-Z] /`?

How can I quote a variable to use in a regex?
 What is `/o` really for?
 How do I use a regular expression to strip C style comments from a file?

Can I use Perl regular expressions to match balanced text?
 What does it mean that regexes are greedy? How can I get around it?
 How do I process each word on each line?
 How can I print out a word–frequency or line–frequency summary?
 How can I do approximate matching?
 How do I efficiently match many regular expressions at once?
 Why don't word–boundary searches with `\b` work for me?
 Why does using `$&`, `$'`, or `$'` slow my program down?
 What good is `\G` in a regular expression?
 Are Perl regexes DFAs or NFAs? Are they POSIX compliant?
 What's wrong with using `grep` or `map` in a void context?
 How can I match strings with multibyte characters?
 How do I match a pattern that is supplied by the user?

AUTHOR AND COPYRIGHT

perlfac7 – General Perl Language Issues (\$Revision: 1.8 \$, \$Date:

2002/03/26 15:48:32 \$)

DESCRIPTION

Can I get a BNF/yacc/RE for the Perl language?
 What are all these `$@%&*` punctuation signs, and how do I know when to use them?

Do I always/never have to quote my strings or use semicolons and commas?

How do I skip some return values?
 How do I temporarily block warnings?
 What's an extension?
 Why do Perl operators have different precedence than C operators?
 How do I declare/create a structure?
 How do I create a module?
 How do I create a class?
 How can I tell if a variable is tainted?
 What's a closure?
 What is variable suicide and how can I prevent it?
 How can I pass/return a {Function, FileHandle, Array, Hash, Method, Regex}?

Passing Variables and Functions, Passing Filehandles, Passing Regexes, Passing Methods

How do I create a static variable?
 What's the difference between dynamic and lexical (static) scoping?
 Between `local()` and `my()`?

How can I access a dynamic variable while a similarly named lexical is in scope?

What's the difference between deep and shallow binding?
 Why doesn't `"my($foo) = <FILE;"` work right?
 How do I redefine a builtin function, operator, or method?

What's the difference between calling a function as `&foo` and `foo()` ?
 How do I create a switch or case statement?
 How can I catch accesses to undefined variables/functions/methods?
 Why can't a method included in this same file be found?
 How can I find out my current package?
 How can I comment out a large block of perl code?
 How do I clear a package?
 How can I use a variable as a variable name?

AUTHOR AND COPYRIGHT

perlfaq8 – System Interaction (\$Revision: 1.8 \$, \$Date: 2002/05/16

12:41:42 \$)

DESCRIPTION

How do I find out which operating system I'm running under?
 How come `exec()` doesn't return?
 How do I do fancy stuff with the keyboard/screen/mouse?
 Keyboard, Screen, Mouse

How do I print something out in color?
 How do I read just one key without waiting for a return key?
 How do I check whether input is ready on the keyboard?
 How do I clear the screen?
 How do I get the screen size?
 How do I ask the user for a password?
 How do I read and write the serial port?
 lockfiles, open mode, end of line, flushing output, non-blocking input

How do I decode encrypted password files?
 How do I start a process in the background?
 STDIN, STDOUT, and STDERR are shared, Signals, Zombies

How do I trap control characters/signals?
 How do I modify the shadow password file on a Unix system?
 How do I set the time and date?
 How can I `sleep()` or `alarm()` for under a second?
 How can I measure time under a second?
 How can I do an `atexit()` or `setjmp()/longjmp()` ? (Exception handling)
 Why doesn't my sockets program work under System V (Solaris)? What
 does the error message "Protocol not supported" mean?

How can I call my system's unique C functions from Perl?
 Where do I get the include files to do `ioctl()` or `syscall()` ?
 Why do setuid perl scripts complain about kernel problems?
 How can I open a pipe both to and from a command?
 Why can't I get the output of a command with `system()` ?
 How can I capture STDERR from an external command?
 Why doesn't `open()` return an error when a pipe open fails?
 What's wrong with using backticks in a void context?
 How can I call backticks without shell processing?
 Why can't my script read from STDIN after I gave it EOF (^D on Unix,
 ^Z on MS-DOS)?

How can I convert my shell script to perl?
 Can I use perl to run a telnet or ftp session?

How can I write expect in Perl?

Is there a way to hide perl's command line from programs such as "ps"?

I {changed directory, modified my environment} in a perl script. How come the change disappeared when I exited the script? How do I get my changes to be visible? Unix

How do I close a process's filehandle without waiting for it to complete?

How do I fork a daemon process?

How do I find out if I'm running interactively or not?

How do I timeout a slow event?

How do I set CPU limits?

How do I avoid zombies on a Unix system?

How do I use an SQL database?

How do I make a `system()` exit on control-C?

How do I open a file without blocking?

How do I install a module from CPAN?

What's the difference between `require` and `use`?

How do I keep my own module/library directory?

How do I add the directory my program lives in to the module/library search path?

How do I add a directory to my include path at runtime?

What is `socket.ph` and where do I get it?

AUTHOR AND COPYRIGHT

perlfaq9 – Networking (\$Revision: 1.9 \$, \$Date: 2002/04/07 18:46:13

\$)

DESCRIPTION

What is the correct form of response from a CGI script?

My CGI script runs from the command line but not the browser. (500 Server Error)

How can I get better error messages from a CGI program?

How do I remove HTML from a string?

How do I extract URLs?

How do I download a file from the user's machine? How do I open a file on another machine?

How do I make a pop-up menu in HTML?

How do I fetch an HTML file?

How do I automate an HTML form submission?

How do I decode or create those %-encodings on the web?

How do I redirect to another page?

How do I put a password on my web pages?

How do I edit my `.htpasswd` and `.htgroup` files with Perl?

How do I make sure users can't enter values into a form that cause my CGI script to do bad things?

How do I parse a mail header?

How do I decode a CGI form?

How do I check a valid mail address?
 How do I decode a MIME/BASE64 string?
 How do I return the user's mail address?
 How do I send mail?
 How do I use MIME to make an attachment to a mail message?
 How do I read mail?
 How do I find out my hostname/domainname/IP address?
 How do I fetch a news article or the active newsgroups?
 How do I fetch/put an FTP file?
 How can I do RPC in Perl?

AUTHOR AND COPYRIGHT

perlcompile – Introduction to the Perl Compiler–Translator

DESCRIPTION

Layout

B::Bytecode, B::C, B::CC, B::Lint, B::Deparse, B::Xref

Using The Back Ends

The Cross Referencing Back End

i, &, s, r

The Decompiling Back End

The Lint Back End

The Simple C Back End

The Bytecode Back End

The Optimized C Back End

Module List for the Compiler Suite

B, O, B::Asmdata, B::Assembler, B::Bblock, B::Bytecode, B::C, B::CC, B::Concise, B::Debug,
 B::Deparse, B::Disassembler, B::Lint, B::Showlex, B::Stackobj, B::Stash, B::Terse, B::Xref

KNOWN PROBLEMS

AUTHOR

perlembed – how to embed perl in your C program

DESCRIPTION

PREAMBLE

Use C from Perl?, Use a Unix program from Perl?, Use Perl from Perl?, Use C from C?,
 Use Perl from C?

ROADMAP

Compiling your C program

Adding a Perl interpreter to your C program

Calling a Perl subroutine from your C program

Evaluating a Perl statement from your C program

Performing Perl pattern matches and substitutions from your C program

Fiddling with the Perl stack from your C program

Maintaining a persistent interpreter

Execution of END blocks

Maintaining multiple interpreter instances

Using Perl modules, which themselves use C libraries, from your C
 program

Embedding Perl under Win32

MORAL

AUTHOR
COPYRIGHT

perldebguts – Guts of Perl debugging

DESCRIPTION

Debugger Internals

Writing Your Own Debugger

Frame Listing Output Examples

Debugging regular expressions

Compile-time output

anchored *STRING* at *POS*, floating *STRING* at *POS1..POS2*, matching
floating/anchored, minlen, stclass *TYPE*, noscan, isall, GPOS, plus,
implicit, with eval, anchored(*TYPE*)

Types of nodes

Run-time output

Debugging Perl memory usage

Using \$ENV{PERL_DEBUG_MSTATS}

buckets SMALLEST (APPROX) .. GREATEST (APPROX), Free/Used, Total sbrk():
SBRKed/SBRKs: CONTINUOUS, pad: 0, heads: 2192, chain: 0, tail: 6144

Example of using **-DL** switch

717, 002, 054, 602, 702, 704

-DL details

!!!, !!, !

Limitations of **-DL** statistics

SEE ALSO

perlxsut, perlXStut – Tutorial for writing XSUBs

DESCRIPTION

SPECIAL NOTES

make

Version caveat

Dynamic Loading versus Static Loading

TUTORIAL

EXAMPLE 1

EXAMPLE 2

What has gone on?

Writing good test scripts

EXAMPLE 3

What's new here?

Input and Output Parameters

The XSUBPP Program

The TYPEMAP file

Warning about Output Arguments

EXAMPLE 4

What has happened here?

Anatomy of .xs file

Getting the fat out of XSUBs

More about XSUB arguments

The Argument Stack

Extending your Extension

- Documenting your Extension
- Installing your Extension
- EXAMPLE 5
- New Things in this Example
- EXAMPLE 6
- New Things in this Example
- EXAMPLE 7 (Coming Soon)
- EXAMPLE 8 (Coming Soon)
- EXAMPLE 9 Passing open files to XSes
- Troubleshooting these Examples
- See also
- Author
- Last Changed

perlxS – XS language reference manual

DESCRIPTION

- Introduction
- On The Road
- The Anatomy of an XSUB
- The Argument Stack
- The RETVAL Variable
- The MODULE Keyword
- The PACKAGE Keyword
- The PREFIX Keyword
- The OUTPUT: Keyword
- The NO_OUTPUT Keyword
- The CODE: Keyword
- The INIT: Keyword
- The NO_INIT Keyword
- Initializing Function Parameters
- Default Parameter Values
- The PREINIT: Keyword
- The SCOPE: Keyword
- The INPUT: Keyword
- The IN/OUTLIST/IN_OUTLIST/OUT/IN_OUT Keywords
- The `length(NAME)` Keyword
- Variable-length Parameter Lists
- The C_ARGS: Keyword
- The PPCODE: Keyword
- Returning Undef And Empty Lists
- The REQUIRE: Keyword
- The CLEANUP: Keyword
- The POSTCALL: Keyword
- The BOOT: Keyword
- The VERSIONCHECK: Keyword
- The PROTOTYPES: Keyword
- The PROTOTYPE: Keyword
- The ALIAS: Keyword
- The OVERLOAD: Keyword
- The INTERFACE: Keyword
- The INTERFACE_MACRO: Keyword
- The INCLUDE: Keyword
- The CASE: Keyword

- The & **Unary Operator**
- Inserting POD, Comments and C Preprocessor Directives
- Using XS With C++
- Interface Strategy
- Perl Objects And C Structures
- The Typemap
- Safely Storing Static Data in XS
 - MY_CXT_KEY, typedef my_cxt_t, START_MY_CXT, MY_CXT_INIT, dMY_CXT, MY_CXT
- EXAMPLES
- XS VERSION
- AUTHOR

perlclib – Internal replacements for standard C library functions

DESCRIPTION

- Conventions
 - t, p, n, s
- File Operations
- File Input and Output
- File Positioning
- Memory Management and String Handling
- Character Class Tests
- stdlib.h** functions
- Miscellaneous functions

SEE ALSO

perlguts – Introduction to the Perl API

DESCRIPTION

Variables

- Datatypes
- What is an "IV"?
- Working with SVs
- Offsets
- What's Really Stored in an SV?
- Working with AVs
- Working with HVs
- Hash API Extensions
- References
- Blessed References and Class Objects
- Creating New Variables
 - GV_ADDMULTI, GV_ADDWARN
- Reference Counts and Mortality
- Stashes and Globs
- Double-Typed SVs
- Magic Variables
- Assigning Magic
- Magic Virtual Tables
- Finding Magic
- Understanding the Magic of Tied Hashes and Arrays
- Localizing changes
 - SAVEINT(int i), SAVEIV(IV i), SAVEI32(I32 i), Savelong(long i), SAVESPTR(s), SAVEPPTR(p), SAVEFREESV(SV *sv), SAVEMORTALIZESV(SV

```

*sv), SAVEFREEOP(OP *op), SAVEFREEPV(p), SAVECLEARSV(SV *sv),
SAVEDDELETE(HV *hv, char *key, I32 length),
SAVEDDESTRUCTOR(DESTRUCTORFUNC_NOCONTEXT_t f, void *p),
SAVEDDESTRUCTOR_X(DESTRUCTORFUNC_t f, void *p), SAVESTACK_POS(), SV*
save_scalar(GV *gv), AV* save_ary(GV *gv), HV* save_hash(GV *gv),
void save_item(SV *item), void save_list(SV **sarg, I32 maxsarg),
SV* save_svref(SV **sptr), void save_aptr(AV **aptr), void
save_hptr(HV **hptr)

```

Subroutines

- XSUBs and the Argument Stack
- Calling Perl Routines from within C Programs
- Memory Allocation
- PerlIO
- Putting a C value on Perl stack
- Scratchpads
- Scratchpads and recursion

Compiled code

- Code tree
- Examining the tree
- Compile pass 1: check routines
- Compile pass 1a: constant folding
- Compile pass 2: context propagation
- Compile pass 3: peephole optimization
- Pluggable runops

Examining internal data structures with the dump functions

How multiple interpreters and concurrency are supported

- Background and PERL_IMPLICIT_CONTEXT
- So what happened to dTHR?
- How do I use all this in extensions?
- Should I do anything special if I call perl from multiple threads?
- Future Plans and PERL_IMPLICIT_SYS

Internal Functions

- A, p, d, s, n, r, f, M, o, j, x
- Formatted Printing of IVs, UVs, and NVs
- Pointer-To-Integer and Integer-To-Pointer
- Source Documentation

Unicode Support

- What **is** Unicode, anyway?
- How can I recognise a UTF8 string?
- How does UTF8 represent Unicode characters?
- How does Perl store UTF8 strings?
- How do I convert a string to UTF8?
- Is there anything else I need to know?

Custom Operators

AUTHORS

SEE ALSO

perlcall – Perl calling conventions from C

DESCRIPTION

An Error Handler, An Event Driven Program

THE CALL_ FUNCTIONS

`call_sv`, `call_pv`, `call_method`, `call_argv`

FLAG VALUES

`G_VOID`

`G_SCALAR`

`G_ARRAY`

`G_DISCARD`

`G_NOARGS`

`G_EVAL`

`G_KEEPPERR`

Determining the Context

KNOWN PROBLEMS

EXAMPLES

No Parameters, Nothing returned

Passing Parameters

Returning a Scalar

Returning a list of values

Returning a list in a scalar context

Returning Data from Perl via the parameter list

Using `G_EVAL`

Using `G_KEEPPERR`

Using `call_sv`

Using `call_argv`

Using `call_method`

Using `GIMME_V`

Using Perl to dispose of temporaries

Strategies for storing Callback Context Information

1. Ignore the problem – Allow only 1 callback,
2. Create a sequence of callbacks – hard wired limit,
3. Use a parameter to map to the Perl callback

Alternate Stack Manipulation

Creating and calling an anonymous subroutine in C

SEE ALSO

AUTHOR

DATE

perlutil – utilities packaged with the Perl distribution

DESCRIPTION

DOCUMENTATION

[*perldoc/perldoc*](#), [*pod2man/pod2man*](#) and [*pod2text/pod2text*](#), [*pod2html/pod2html*](#) and [*pod2latex/pod2latex*](#), [*pod2usage/pod2usage*](#), [*podselect/podselect*](#), [*podchecker/podchecker*](#), [*splain/splain*](#), [*roffitall/roffitall*](#)

CONVERTORS

[*a2p/a2p*](#), [*s2p/s2p*](#), [*find2perl/find2perl*](#)

Administration

[*libnetcfg/libnetcfg*](#)

Development

[*perlbug/perlbug*](#), [*h2ph/h2ph*](#), [*c2ph/c2ph*](#) and [*pstruct/pstruct*](#), [*h2xs/h2xs*](#), [*dprofpp/dprofpp*](#), [*perlcc/perlcc*](#)

SEE ALSO

perlfiter – Source Filters

DESCRIPTION
CONCEPTS
USING FILTERS
WRITING A SOURCE FILTER
WRITING A SOURCE FILTER IN C

Decryption Filters

CREATING A SOURCE FILTER AS A SEPARATE EXECUTABLE
WRITING A SOURCE FILTER IN PERL
USING CONTEXT: THE DEBUG FILTER
CONCLUSION
REQUIREMENTS
AUTHOR
Copyrights

perlDbmfilter – Perl DBM Filters

SYNOPSIS
DESCRIPTION

filter_store_key, filter_store_value, filter_fetch_key, filter_fetch_value

The Filter

An Example — the NULL termination problem.

Another Example — Key is a C int.

SEE ALSO

AUTHOR

perlapi – autogenerated documentation for the perl public API

DESCRIPTION
"Gimme" Values

GIMME, GIMME_V, G_ARRAY, G_DISCARD, G_EVAL, G_NOARGS, G_SCALAR, G_VOID

Array Manipulation Functions

AvFILL, av_clear, av_delete, av_exists, av_extend, av_fetch, av_fill, av_len, av_make, av_pop, av_push, av_shift, av_store, av_undef, av_unshift, get_av, newAV, Nullav, sortsv

Callback Functions

call_argv, call_method, call_pv, call_sv, ENTER, eval_pv, eval_sv, FREETMPS, LEAVE, SAVETMPS

Character classes

isALNUM, isALPHA, isDIGIT, isLOWER, isSPACE, isUPPER, toLOWER, toUPPER

Cloning an interpreter

perl_clone

CV Manipulation Functions

CvSTASH, get_cv, Nullcv

Embedding Functions

load_module, nothreadhook, perl_alloc, perl_construct, perl_destruct, perl_free, perl_parse, perl_run, require_pv

Functions in file pp_pack.c

pack_cat, unpack_str

Global Variables

PL_modglobal, PL_na, PL_sv_no, PL_sv_undef, PL_sv_yes

GV Functions

GvSV, gv_fetchmeth, gv_fetchmethod, gv_fetchmethod_autoload, gv_fetchmeth_autoload, gv_stashpv, gv_stashsv

Handy Values

HEf_SVKEY, Nullch, Nullsv

Hash Manipulation Functions

get_hv, HeHASH, HeKEY, HeKLEN, HePV, HeSVKEY, HeSVKEY_force, HeSVKEY_set, HeVAL, HvNAME, hv_clear, hv_delete, hv_delete_ent, hv_exists, hv_exists_ent, hv_fetch, hv_fetch_ent, hv_iterinit, hv_iterkey, hv_iterkeysv, hv_itternext, hv_itternextsv, hv_itternext_flags, hv_interval, hv_magic, hv_store, hv_store_ent, hv_undef, newHV, Nullhv

Magical Functions

mg_clear, mg_copy, mg_find, mg_free, mg_get, mg_length, mg_magical, mg_set, SvGETMAGIC, SvLOCK, SvSETMAGIC, SvSetMagicSV, SvSetMagicSV_nosteal, SvSetSV, SvSetSV_nosteal, SvSHARE

Memory Management

Copy, Move, New, Newc, NEWSV, Newz, Poison, Renew, Renewc, Safefree, savepv, savepvn, savesharedpv, StructCopy, Zero

Miscellaneous Functions

fbm_compile, fbm_instr, form, getcwd_sv, strEQ, strGE, strGT, strLE, strLT, strNE, strnEQ, strnNE

Numeric functions

grok_bin, grok_hex, grok_number, grok_numeric_radix, grok_oct, scan_bin, scan_hex, scan_oct

Optree Manipulation Functions

cv_const_sv, newCONSTSUB, newXS

Stack Manipulation Macros

dMARK, dORIGMARK, dSP, EXTEND, MARK, ORIGMARK, POPI, POPI, POPn, POPp, POPpbytex, POPpx, POPs, PUSHi, PUSHMARK, PUSHn, PUSHp, PUSHs, PUSHu, PUTBACK, SP, SPAGAIN, XPUSHi, XPUSHn, XPUSHp, XPUSHs, XPUSHu, XSRETURN, XSRETURN_IV, XSRETURN_NO, XSRETURN_NV, XSRETURN_PV, XSRETURN_UNDEF, XSRETURN_YES, XST_mIV, XST_mNO, XST_mNV, XST_mPV, XST_mUNDEF, XST_mYES

SV Flags

svtype, SVt_IV, SVt_NV, SVt_PV, SVt_PVAV, SVt_PVCV, SVt_PVHV, SVt_PVMG

SV Manipulation Functions

get_sv, looks_like_number, newRV_inc, newRV_noinc, newSV, newSViv, newSVnv, newSVpv, newSVpvf, newSVpvn, newSVpvn_share, newSVrv, newSVsv, newSVuv, new_vstring, SvCUR, SvCUR_set, SvEND, SvGROW, SvIOK, SvIOKp, SvIOK_notUV, SvIOK_off, SvIOK_on, SvIOK_only, SvIOK_only_UV, SvIOK_UV, SvIV, SvIVx, SvIVX, SvLEN, SvNIOK, SvNIOKp, SvNIOK_off, SvNOK, SvNOKp, SvNOK_off, SvNOK_on, SvNOK_only, SvNV, SvNVX, SvNVx, SvOK, SvOOK, SvPOK, SvPOKp, SvPOK_off, SvPOK_on, SvPOK_only, SvPOK_only_UTF8, SvPV, SvPVbyte, SvPVbytex, SvPVbytex_force, SvPVbyte_force, SvPVbyte_nolen, SvPVutf8, SvPVutf8x, SvPVutf8x_force, SvPVutf8_force, SvPVutf8_nolen, SvPVx, SvPVX, SvPV_force, SvPV_force_nomg, SvPV_nolen, SvREFCNT, SvREFCNT_dec, SvREFCNT_inc, SvROK,

SvROK_off, SvROK_on, SvRV, SvSTASH, SvTAINT, SvTAINTED, SvTAINTED_off, SvTAINTED_on, SvTRUE, SvTYPE, SvUNLOCK, SvUOK, SvUPGRADE, SvUTF8, SvUTF8_off, SvUTF8_on, SvUV, SvUVX, SvUVx, sv_2bool, sv_2cv, sv_2io, sv_2iv, sv_2mortal, sv_2nv, sv_2pvbyte, sv_2pvbyte_nolen, sv_2pvutf8, sv_2pvutf8_nolen, sv_2pv_flags, sv_2pv_nolen, sv_2uv, sv_backoff, sv_bless, sv_catpv, sv_catpvf, sv_catpvf_mg, sv_catpv_nolen, sv_catpv_nolen, sv_catpv_mg, sv_catsv, sv_catsv_flags, sv_catsv_mg, sv_chop, sv_clear, sv_cmp, sv_cmp_locale, sv_collxfrm, sv_copypv, sv_dec, sv_derived_from, sv_eq, sv_force_normal, sv_force_normal_flags, sv_free, sv_gets, sv_grow, sv_inc, sv_insert, sv_isa, sv_isobject, sv_iv, sv_len, sv_len_utf8, sv_magic, sv_magicext, sv_mortalcopy, sv_newmortal, sv_newref, sv_nolocking, sv_nosharing, sv_nounlocking, sv_nv, sv_pos_b2u, sv_pos_u2b, sv_pv, sv_pvbyte, sv_pvbyten, sv_pvbyten_force, sv_pvn, sv_pvn_force, sv_pvn_force_flags, sv_pvutf8, sv_pvutf8n, sv_pvutf8n_force, sv_reftype, sv_replace, sv_report_used, sv_reset, sv_rvweaken, sv_setiv, sv_setiv_mg, sv_setnv, sv_setnv_mg, sv_setpv, sv_setpvf, sv_setpvf_mg, sv_setpvn, sv_setpvn_mg, sv_setpv_mg, sv_setref_iv, sv_setref_nv, sv_setref_pv, sv_setref_pvn, sv_setref_uv, sv_setsv, sv_setsv_flags, sv_setsv_mg, sv_setuv, sv_setuv_mg, sv_taint, sv_tainted, sv_true, sv_unmagic, sv_unref, sv_unref_flags, sv_untaint, sv_upgrade, sv_usepvn, sv_usepvn_mg, sv_utf8_decode, sv_utf8_downgrade, sv_utf8_encode, sv_utf8_upgrade, sv_utf8_upgrade_flags, sv_uv, sv_vcatpvfn, sv_vsetpvfn

Unicode Support

bytes_from_utf8, bytes_to_utf8, ibcmp_utf8, is_utf8_char, is_utf8_string, pv_uni_display, sv_recode_to_utf8, sv_uni_display, to_utf8_case, to_utf8_fold, to_utf8_lower, to_utf8_title, to_utf8_upper, utf8n_to_uvchr, utf8n_to_uvuni, utf8_distance, utf8_hop, utf8_length, utf8_to_bytes, utf8_to_uvchr, utf8_to_uvuni, uvchr_to_utf8, uvuni_to_utf8_flags

Variables created by xsubpp and xsubpp internal functions

ax, CLASS, dAX, dITEMS, dXSARGS, dXSI32, items, ix, newXSproto, RETVAL, ST, THIS, XS, XSRETURN_EMPTY, XS_VERSION, XS_VERSION_BOOTCHECK

Warning and Dieing

croak, warn

AUTHORS

SEE ALSO

perlintern – autogenerated documentation of purely internal

Perl functions

DESCRIPTION

Global Variables

PL_DBsingle, PL_DBsub, PL_DBtrace, PL_dowarn, PL_last_in_gv, PL_ofs_sv, PL_rs

GV Functions

is_gv_magical

IO Functions

start_glob

Pad Data Structures

CvPADLIST

Stack Manipulation Macros

djSP, LVRET

SV Manipulation Functions

report_uninit, sv_add_arena, sv_clean_all, sv_clean_objs, sv_free_arenas

AUTHORS
SEE ALSO

perliol – C API for Perl's implementation of IO in Layers.

SYNOPSIS

DESCRIPTION

History and Background
Layers vs Disciplines
Data Structures
Functions and Attributes
Per-instance Data
Layers in action.
Per-instance flag bits

PERLIO_F_EOF, PERLIO_F_CANWRITE, PERLIO_F_CANREAD, PERLIO_F_ERROR,
PERLIO_F_TRUNCATE, PERLIO_F_APPEND, PERLIO_F_CRLF, PERLIO_F_UTF8,
PERLIO_F_UNBUF, PERLIO_F_WRBUF, PERLIO_F_RDBUF, PERLIO_F_LINEBUF,
PERLIO_F_TEMP, PERLIO_F_OPEN, PERLIO_F_FASTGETS

Methods in Detail

fsize, name, size, kind, PERLIO_K_BUFFERED, PERLIO_K_RAW, PERLIO_K_CANCRRLF,
PERLIO_K_FASTGETS, PERLIO_K_MULTIARG, Pushed, Popped, Open, Binmode, Getarg,
Fileno, Dup, Read, Write, Seek, Tell, Close, Flush, Fill, Eof, Error, Clearerr, Setlinebuf,
Get_base, Get_bufsiz, Get_ptr, Get_cnt, Set_ptrcnt

Core Layers

"unix", "perlio", "stdio", "crlf", "mmap", "pending", "raw", "utf8"

Extension Layers

":encoding", ":scalar", ":via"

TODO

perlapi – perl's IO abstraction interface.

SYNOPSIS

DESCRIPTION

1. USE_STDIO, 2. USE_SFIO, 3. USE_PERLIO, `PerlIO_stdin()`, `PerlIO_stdout()`,
`PerlIO_stderr()`, `PerlIO_open(path, mode)`, `PerlIO_fdopen(fd, mode)`,
`PerlIO_reopen(path, mode, f)`, `PerlIO_printf(f, fmt, ...)`, `PerlIO_vprintf(f, fmt, a)`,
`PerlIO_stdoutf(fmt, ...)`, `PerlIO_read(f, buf, count)`, `PerlIO_write(f, buf, count)`, `PerlIO_close(f)`,
`PerlIO_puts(f, s)`, `PerlIO_putc(f, c)`, `PerlIO_ungetc(f, c)`, `PerlIO_getc(f)`, `PerlIO_eof(f)`,
`PerlIO_error(f)`, `PerlIO_fileno(f)`, `PerlIO_clearerr(f)`, `PerlIO_flush(f)`,
`PerlIO_seek(f, offset, whence)`, `PerlIO_tell(f)`, `PerlIO_getpos(f, p)`, `PerlIO_setpos(f, p)`,
`PerlIO_rewind(f)`, `PerlIO_tmpfile()`, `PerlIO_setlinebuf(f)`

Co-existence with stdio

`PerlIO_importFILE(f, mode)`, `PerlIO_exportFILE(f, mode)`, `PerlIO_releaseFILE(p, f)`,
`PerlIO_findFILE(f)`

"Fast gets" Functions

`PerlIO_fast_gets(f)`, `PerlIO_has_cntptr(f)`, `PerlIO_get_cnt(f)`, `PerlIO_get_ptr(f)`,
`PerlIO_set_ptrcnt(f, p, c)`, `PerlIO_canset_cnt(f)`, `PerlIO_set_cnt(f, c)`, `PerlIO_has_base(f)`,
`PerlIO_get_base(f)`, `PerlIO_get_bufsiz(f)`

Other Functions

`PerlIO_apply_layers(f, mode, layers)`, `PerlIO_binmode(f, ptype, imode, layers)`, '<' read, '>' write,
'+' read/write, `PerlIO_debug(fmt, ...)`

perltodo – Perl TO–DO List**DESCRIPTION****To do during 5.6.x**

- Support for I/O disciplines
- Autoload bytes.pm
- Make "\u{XXXX}" et al work
- Create a char *sv_pvprintify(sv, STRLEN *lenp, UV flags)
- Overloadable regex assertions
- Unicode
- Work out exit/die semantics for threads
- Better support for nonpreemptive threading systems like GNU pth
- Typed lexicals for compiler
- Compiler workarounds for Win32
- AUTOLOADing in the compiler
- Fixing comppadlist when compiling
- Cleaning up exported namespace
- Complete signal handling
- Out-of-source builds
- POSIX realtime support
- UNIX98 support
- IPv6 Support
- Long double conversion
- Locales
- Arithmetic on non-Arabic numerals
- POSIX Unicode character classes
- Factoring out common suffices/prefices in regexps (trie optimization)
- Security audit shipped utilities
- Sort out the uid-setting mess
- Custom opcodes
- DLL Versioning
- Introduce @(and @)
- Floating point handling
- IV/UV preservation
- Replace pod2html with something using Pod::Parser
- Automate module testing on CPAN
- sendmsg and recvmsg
- Rewrite perlre documentation
- Convert example code to IO::Handle filehandles
- Document Win32 choices
- Check new modules
- Make roffitall find pods and libs itself

To do at some point

- Remove regular expression recursion
- Memory leaks after failed eval
- bitfields in pack
- Cross compilation
- Perl preprocessor / macros
- Perl lexer in Perl
- Using POSIX calls internally
- i rename file when changed
- All ARGV input should act like <>
- Support for rerunning debugger

- Test Suite for the Debugger
- my sub foo { }
- One-pass global destruction
- Rewrite regexp parser
- Cache recently used regexps
- Cross-compilation support
- Bit-shifting bitvectors
- debugger pragma
- use less pragma
- switch structures
- Cache eval tree
- rcatmaybe
- Shrink opcode tables
- Optimize away @_
- Prototypes versus indirect objects
- Install HTML
- Prototype method calls
- Return context prototype declarations
- magic_setisa
- Garbage collection
- IO tutorial
- Rewrite perldoc
- Install .3p manpages
- Unicode tutorial
- Update POSIX.pm for 1003.1-2
- Retargetable installation
- POSIX emulation on non-POSIX systems
- Rename Win32 headers
- Finish off lvalue functions
- Update sprintf documentation
- Use fchown/fchmod internally
- Make v-strings overloaded objects
- Allow restricted hash assignment
- Should overload be inheritable?
- Taint rethink
- Vague ideas
 - ref() in list context
 - Make tr/// return histogram of characters in list context
 - Compile to real threaded code
 - Structured types
 - Modifiable \$! et al.
 - Procedural interfaces for IO::*, etc.
 - RPC modules
 - Attach/detach debugger from running program
 - GUI::Native
 - foreach(reverse ...)
 - Constant function cache
 - Approximate regular expression matching
- Ongoing
 - Update guts documentation
 - Add more tests
 - Update auxiliary tools
 - Create debugging macros

- truncate to the people
- Unicode in Filenames
- Recently done things
 - Alternative RE syntax module
 - Safe signal handling
 - Tie Modules
 - gettimeofday
 - setitimer and getitimer
 - Testing `__DIE__` hook
 - CPP equivalent in Perl
 - Explicit switch statements
 - autocroak
 - UTF/EBCDIC
 - UTF Regexes
 - perlcc to produce executable
 - END blocks saved in compiled output
 - Secure temporary file module
 - Integrate Time::HiRes
 - Turn Cwd into XS
 - Mmap for input
 - Byte to/from UTF8 and UTF8 to/from local conversion
 - Add socketmark support
 - Mailing list archives
 - Bug tracking
 - Integrate MacPerl
 - Web "nerve center" for Perl
 - Regular expression tutorial
 - Debugging Tutorial
 - Integrate new modules
 - Integrate profiler
 - Y2K error detection
 - Regular expression debugger
 - POD checker
 - "Dynamic" lexicals
 - Cache precompiled modules
- Deprecated Wishes
 - Loop control on `do{}`
 - Lexically scoped typeglobs
 - format BOTTOM
 - report HANDLE
 - Generalised `want()/caller()`
 - Named prototypes
 - Built-in globbing
 - Regression tests for `suidperl`
 - Cached hash values
 - Add compression modules
 - Reorganise documentation into tutorials/references
 - Remove distinction between functions and operators
 - Make XS easier to use
 - Make embedding easier to use
 - man for perl
 - `my $Package::variable`
 - "or" tests defined, not truth

"class"-based lexicals
byterperl
Lazy evaluation / tail recursion removal
Make "use utf8" the default
Unicode collation and normalization
pack/unpack tutorial

perlhack – How to hack at the Perl internals

DESCRIPTION

Does concept match the general goals of Perl?, Where is the implementation?, Backwards compatibility, Could it be a module instead?, Is the feature generic enough?, Does it potentially introduce new bugs?, Does it preclude other desirable features?, Is the implementation robust?, Is the implementation generic enough to be portable?, Is the implementation tested?, Is there enough documentation?, Is there another way to do it?, Does it create too much work?, Patches speak louder than words

Keeping in sync

rsync'ing the source tree, Using rsync over the LAN, Using pushing over the NFS, rsync'ing the patches

Why rsync the source tree

It's easier to rsync the source tree, It's more reliable

Why rsync the patches

It's easier to rsync the patches, It's a good reference, Finding a start point, Finding how to fix a bug, Finding the source of misbehaviour

Perlbug remote interface

1 <http://bugs.perl.org>, 2 bugdb@perl.org, 3 commands_and_bugdids@bugs.perl.org, notes, patches, tests

Submitting patches

[perlguts](#), [perlxtut](#) and [perlxs](#), [perlapi](#), *Porting/pumpkin.pod*, The perl5-porters FAQ

Finding Your Way Around

Core modules, Tests, Documentation, Configure, Interpreter

Elements of the interpreter

Startup, Parsing, Optimization, Running

Internal Variable Types

Op Trees

Stacks

Argument stack, Mark stack, Save stack

Millions of Macros

Poking at Perl

Using a source-level debugger

run [args], break function_name, break source.c:xxx, step, next, continue, finish, 'enter', print

Dumping Perl Data Structures

Patching

Patching a core module

Adding a new function to the core

Writing a test

t/base/, *t/cmd/*, *t/comp/*, *t/io/*, *t/lib/*, *t/op/*, *t/pod/*, *t/run/*, *t/base* *t/comp*, *t/cmd* *t/run* *t/io* *t/op*, *t/lib*
ext lib

Special Make Test Targets

coretest, test.deparse, minitest, test.third check.third utest.third ucheck.third, test.torture
 torturetest, utest ucheck test.utf8 check.utf8

EXTERNAL TOOLS FOR DEBUGGING PERL

Rational Software's Purify

Purify on Unix

-Accflags=-DPURIFY, -Doptimize='-g', -Uusemymalloc, -Dusemultiplicity

Purify on NT

DEFINES, USE_MULTI = define, #PERL_MALLOC = define, CFG = Debug

Compaq's/Digital's/HP's Third Degree

PERL_DESTRUCT_LEVEL

Profiling

Gprof Profiling

-a, -b, -e routine, -f routine, -s, -z

GCC gcov Profiling

Pixie Profiling

-h, -l, -p[rocedures], -h[eavy], -i[nvocations], -l[ines], -testcoverage, -z[ero]

Miscellaneous tricks

CONCLUSION

The Road goes ever on and on, down from the door where it began.

AUTHOR

perlhist – the Perl history records

DESCRIPTION

INTRODUCTION

THE KEEPERS OF THE PUMPKIN

PUMPKIN?

THE RECORDS

SELECTED RELEASE SIZES

SELECTED PATCH SIZES

THE KEEPERS OF THE RECORDS

perldelta – what is new for perl v5.8.0

DESCRIPTION

Highlights In 5.8.0

Incompatible Changes

Binary Incompatibility

64-bit platforms and malloc

AIX Dynaloading

Attributes for `my` variables now handled at run-time

Socket Extension Dynamic in VMS

IEEE-format Floating Point Default on OpenVMS Alpha

New Unicode Semantics (no more `use utf8`, almost)

New Unicode Properties

`REF(...)` Instead Of `SCALAR(...)`

`pack/unpack D/F` recycled

`glob()` now returns filenames in alphabetical order

Deprecations

- Core Enhancements
 - Unicode Overhaul
 - PerlIO is Now The Default
 - ithreads
 - Restricted Hashes
 - Safe Signals
 - Understanding of Numbers
 - Arrays now always interpolate into double-quoted strings [561]
 - Miscellaneous Changes
- Modules and Pragmata
 - New Modules and Pragmata
 - Updated And Improved Modules and Pragmata
- Utility Changes
- New Documentation
- Performance Enhancements
- Installation and Configuration Improvements
 - Generic Improvements
 - New Or Improved Platforms
- Selected Bug Fixes
 - Platform Specific Changes and Fixes
- New or Changed Diagnostics
- Changed Internals
- Security Vulnerability Closed [561]
- New Tests
- Known Problems
 - The Compiler Suite Is Still Very Experimental
 - Localising Tied Arrays and Hashes Is Broken
 - Building Extensions Can Fail Because Of Largefiles
 - Modifying \$_ Inside for(..)
 - mod_perl 1.26 Doesn't Build With Threaded Perl
 - lib/ftmp-security tests warn 'system possibly insecure'
 - libwww-perl (LWP) fails base/date #51
 - PDL failing some tests
 - Perl_get_sv
 - Self-tying Problems
 - ext/threads/t/libc
 - Failure of Thread (5.005-style) tests
 - Timing problems
 - Tied/Magical Array/Hash Elements Do Not Autovivify
 - Unicode in package/class and subroutine names does not work
- Platform Specific Problems
 - AIX
 - Alpha systems with old gccs fail several tests
 - AmigaOS
 - BeOS
 - Cygwin "unable to remap"
 - Cygwin ndbm tests fail on FAT
 - DJGPP Failures
 - FreeBSD built with ithreads coredumps reading large directories
 - FreeBSD Failing locale Test 117 For ISO 8859-15 Locales
 - IRIX fails ext/List/Util/t/shuffle.t or Digest::MD5
 - HP-UX lib/posix Subtest 9 Fails When LP64-Configured
 - Linux with glibc 2.2.5 fails t/op/int subtest #6 with -Duse64bitint

- Linux With Sdio Fails op/misc Test 48
- Mac OS X
- Mac OS X dyld undefined symbols
- OS/2 Test Failures
- op/sprintf tests 91, 129, and 130
- Solaris 2.5
- Solaris x86 Fails Tests With -Duse64bitint
- SUPER-UX (NEC SX)
- Term::ReadKey not working on Win32
- UNICOS/mk
- UTS
- VOS (Stratus)
- VMS
- Win32
- XML::Parser not working
- z/OS (OS/390)
- Unicode Support on EBCDIC Still Spotty
- Seen In Perl 5.7 But Gone Now
- Reporting Bugs
- SEE ALSO
- HISTORY

perl572delta – what’s new for perl v5.7.2

- DESCRIPTION
- Security Vulnerability Closed
- Incompatible Changes
 - 64-bit platforms and malloc
 - AIX Dynaloading
 - Socket Extension Dynamic in VMS
 - Different Definition of the Unicode Character Classes \p{In...}
- Deprecations
- Core Enhancements
- Modules and Pragmata
 - New Modules and Distributions
 - Updated And Improved Modules and Pragmata
- Utility Changes
- New Documentation
- Installation and Configuration Improvements
 - New Or Improved Platforms
 - Generic Improvements
- Selected Bug Fixes
 - Platform Specific Changes and Fixes
- New or Changed Diagnostics
- Source Code Enhancements
 - MAGIC constants
 - Better commented code
 - Regex pre-/post-compilation items matched up
 - gcc -Wall
- New Tests
- Known Problems
 - AIX
 - Amiga Perl Invoking Mystery
 - lib/ftmp-security tests warn ‘system possibly insecure’

Cygwin intermittent failures of lib/Memoize/t/expire_file 11 and 12
HP-UX lib/io_multihomed Fails When LP64-Configured
HP-UX lib/posix Subtest 9 Fails When LP64-Configured
Linux With Sflo Fails op/misc Test 48
OS/390
op/sprintf tests 129 and 130
Failure of Thread tests
UNICOS
UTS
VMS
Win32
Localising a Tied Variable Leaks Memory
Self-tying of Arrays and Hashes Is Forbidden
Variable Attributes are not Currently Usable for Tieing
Building Extensions Can Fail Because Of Largefiles
The Compiler Suite Is Still Experimental
The Long Double Support is Still Experimental
Reporting Bugs
SEE ALSO
HISTORY

perl571delta – what's new for perl v5.7.1

DESCRIPTION

Security Vulnerability Closed

Incompatible Changes

Core Enhancements

AUTOLOAD Is Now Lvaluable

PerlIO is Now The Default

Signals Are Now Safe

Modules and Pragmata

New Modules

Updated And Improved Modules and Pragmata

Performance Enhancements

Utility Changes

New Documentation

perlclib

perliol

README.aix

README.bs2000

README.macos

README.mpeix

README.solaris

README.vos

Porting/repository.pod

Installation and Configuration Improvements

New Or Improved Platforms

Generic Improvements

d_cmsg_hdr, d_fcntl_can_lock, d_fsync, d_getitimer, d_getpagsz, d_msg_hdr_s, need_va_copy,
d_readv, d_recvmsg, d_sendmsg, sig_size, d_socketmark, d_strtoq, d_u32align, d_ualarm,
d_usleep

Selected Bug Fixes

Platform Specific Changes and Fixes

New or Changed Diagnostics

Changed Internals

New Tests

Known Problems

- AIX vac 5.0.0.0 May Produce Buggy Code For Perl

- lib/ftmp–security tests warn ‘system possibly insecure’

- lib/io_multihomed Fails In LP64–Configured HP–UX

- Test lib/posix Subtest 9 Fails In LP64–Configured HP–UX

- lib/b test 19

- Linux With Sflo Fails op/misc Test 48

- sigaction test 13 in VMS

- sprintf tests 129 and 130

- Failure of Thread tests

- Localising a Tied Variable Leaks Memory

- Self–tying of Arrays and Hashes Is Forbidden

- Building Extensions Can Fail Because Of Largefiles

- The Compiler Suite Is Still Experimental

Reporting Bugs

SEE ALSO

HISTORY

perl570delta – what’s new for perl v5.7.0

DESCRIPTION

Security Vulnerability Closed

Incompatible Changes

Core Enhancements

Modules and Pragmata

- New Modules

- Updated And Improved Modules and Pragmata

Utility Changes

New Documentation

Performance Enhancements

Installation and Configuration Improvements

- Generic Improvements

Selected Bug Fixes

- Platform Specific Changes and Fixes

New or Changed Diagnostics

Changed Internals

Known Problems

- Unicode Support Still Far From Perfect

- EBCDIC Still A Lost Platform

- Building Extensions Can Fail Because Of Largefiles

- ftmp–security tests warn ‘system possibly insecure’

- Test lib/posix Subtest 9 Fails In LP64–Configured HP–UX

- Long Doubles Still Don’t Work In Solaris

- Linux With Sflo Fails op/misc Test 48

- Storable tests fail in some platforms

- Threads Are Still Experimental

- The Compiler Suite Is Still Experimental

Reporting Bugs

SEE ALSO

HISTORY

perl561delta – what's new for perl v5.6.x**DESCRIPTION**

Summary of changes between 5.6.0 and 5.6.1

Security Issues

Core bug fixes

UNIVERSAL::isa(), Memory leaks, Numeric conversions, qw(a\\b), caller(), Bugs in regular expressions, "slurp" mode, Autovivification of symbolic references to special variables, Lexical warnings, Spurious warnings and errors, glob(), Tainting, sort(), #line directives, Subroutine prototypes, map(), Debugger, PERL5OPT, chop(), Unicode support, 64-bit support, Compiler, Lvalue subroutines, IO::Socket, File::Find, xsubpp, no Module;, Tests

Core features

Configuration issues

Documentation

Bundled modules

B::Concise, File::Temp, Pod::LaTeX, Pod::Text::Overstrike, CGI, CPAN, Class::Struct, DB_File, Devel::Peek, File::Find, Getopt::Long, IO::Poll, IPC::Open3, Math::BigFloat, Math::Complex, Net::Ping, Opcode, Pod::Parser, Pod::Text, SDBM_File, Sys::Syslog, Tie::RefHash, Tie::SubstrHash

Platform-specific improvements

NCR MP-RAS, NonStop-UX

Core Enhancements

Interpreter cloning, threads, and concurrency

Lexically scoped warning categories

Unicode and UTF-8 support

Support for interpolating named characters

"our" declarations

Support for strings represented as a vector of ordinals

Improved Perl version numbering system

New syntax for declaring subroutine attributes

File and directory handles can be autovivified

open() with more than two arguments

64-bit support

Large file support

Long doubles

"more bits"

Enhanced support for sort() subroutines

sort \$coderef @foo allowed

File globbing implemented internally

Support for CHECK blocks

POSIX character class syntax [: :] supported

Better pseudo-random number generator

Improved qw// operator

Better worst-case behavior of hashes

pack() format 'Z' supported

pack() format modifier '!' supported

pack() and unpack() support counted strings

Comments in pack() templates

Weak references

Binary numbers supported

Lvalue subroutines

Some arrows may be omitted in calls through references
 Boolean assignment operators are legal lvalues
`exists()` is supported on subroutine names
`exists()` and `delete()` are supported on array elements
 Pseudo-hashes work better
 Automatic flushing of output buffers
 Better diagnostics on meaningless filehandle operations
 Where possible, buffered data discarded from duped input filehandle
`eof()` has the same old magic as `<`
`binmode()` can be used to set `:crlf` and `:raw` modes
`-T` filetest recognizes UTF-8 encoded files as "text"
`system()`, backticks and pipe open now reflect `exec()` failure
 Improved diagnostics
 Diagnostics follow `STDERR`
 More consistent close-on-exec behavior
`syswrite()` ease-of-use
 Better syntax checks on parenthesized unary operators
 Bit operators support full native integer width
 Improved security features
 More functional bareword prototype (*)
`require` and `do` may be overridden
`$^X` variables may now have names longer than one character
 New variable `$^C` reflects `-c` switch
 New variable `$^V` contains Perl version as a string
 Optional Y2K warnings
 Arrays now always interpolate into double-quoted strings
 Modules and Pragmata
 Modules
 `attributes`, `B`, `Benchmark`, `ByteLoader`, `constant`, `chardnames`, `Data::Dumper`, `DB`, `DB_File`,
 `Devel::DProf`, `Devel::Peek`, `Dumpvalue`, `DynaLoader`, `English`, `Env`, `Fcntl`, `File::Compare`,
 `File::Find`, `File::Glob`, `File::Spec`, `File::Spec::Functions`, `Getopt::Long`, `IO`, `JPL`, `lib`,
 `Math::BigInt`, `Math::Complex`, `Math::Trig`, `Pod::Parser`, `Pod::InputObjects`, `Pod::Checker`,
 `podchecker`, `Pod::ParseUtils`, `Pod::Find`, `Pod::Select`, `podselect`, `Pod::Usage`, `pod2usage`,
 `Pod::Text` and `Pod::Man`, `SDBM_File`, `Sys::Syslog`, `Sys::Hostname`, `Term::ANSIColor`,
 `Time::Local`, `Win32`, `XSLoader`, `DBM Filters`
 Pragmata
 Utility Changes
 `dprofpp`
 `find2perl`
 `h2xs`
 `perlcc`
 `perldoc`
 The Perl Debugger
 Improved Documentation
 `perlapi.pod`, `perlboot.pod`, `perlcompile.pod`, `perldbfilter.pod`, `perldebug.pod`, `perldebbugs.pod`,
 `perlfork.pod`, `perlfiter.pod`, `perlhack.pod`, `perlintern.pod`, `perllexwarn.pod`, `perlnumber.pod`,
 `perlopentut.pod`, `perlreftut.pod`, `perltootc.pod`, `perltodo.pod`, `perlunicode.pod`
 Performance enhancements
 Simple `sort()` using `{ $a <= $b }` and the like are optimized
 Optimized assignments to lexical variables
 Faster subroutine calls

- delete(), each(), values() and hash iteration are faster**
- Installation and Configuration Improvements
 - Dusethreads means something different
 - New Configure flags
 - Threadedness and 64-bitness now more daring
 - Long Doubles
 - Dusemorebits
 - Duselargefiles
 - installusrbinperl
 - SOCKS support
 - A flag
 - Enhanced Installation Directories
 - gcc automatically tried if ‘cc’ does not seem to be working
- Platform specific changes
 - Supported platforms
 - DOS
 - OS390 (OpenEdition MVS)
 - VMS
 - Win32
- Significant bug fixes
 - <HANDLE on empty files
 - eval ‘...’ improvements
 - All compilation errors are true errors
 - Implicitly closed filehandles are safer
 - Behavior of list slices is more consistent
 - (\ \$) prototype and \$foo{a}
 - goto &sub and AUTOLOAD
 - bareword allowed under use integer
 - Failures in DESTROY()
 - Locale bugs fixed
 - Memory leaks
 - Spurious subroutine stubs after failed subroutine calls
 - Taint failures under –U
 - END blocks and the –c switch
 - Potential to leak DATA filehandles
- New or Changed Diagnostics
 - %s" variable %s masks earlier declaration in same %s, "my sub" not yet implemented, "our" variable %s redeclared, ‘!’ allowed only after types %s, / cannot take a count, / must be followed by a, A or Z, / must be followed by a*, A* or Z*, / must follow a numeric type, /%s/: Unrecognized escape \\%c passed through, /%s/: Unrecognized escape \\%c in character class passed through, /%s/ should probably be written as "%s", %s() called too early to check prototype, %s argument is not a HASH or ARRAY element, %s argument is not a HASH or ARRAY element or slice, %s argument is not a subroutine name, %s package attribute may clash with future reserved word: %s, (in cleanup) %s, < should be quotes, Attempt to join self, Bad evalled substitution pattern, Bad realloc() ignored, Bareword found in conditional, Binary number 0b11111111111111111111111111111111 non-portable, Bit vector size 32 non-portable, Buffer overflow in prime_env_iter: %s, Can't check filesystem of script "%s", Can't declare class for non-scalar %s in "%s", Can't declare %s in "%s", Can't ignore signal CHLD, forcing to default, Can't modify non-lvalue subroutine call, Can't read CRTL environ, Can't remove %s: %s, skipping file, Can't return %s from lvalue subroutine, Can't weaken a nonreference, Character class [:%s:] unknown, Character class syntax [%s] belongs inside character classes, Constant is not %s reference, constant(%s): %s, CORE::%s is not a keyword, defined(@array) is deprecated, defined(%hash) is deprecated, Did not produce a valid header, (Did you mean "local" instead of "our"?), Document contains no data, entering effective %s failed, false [] range "%s" in regexp, Filehandle %s opened only for output, flock() on closed filehandle %s,

Global symbol "%s" requires explicit package name, Hexadecimal number 0xffffffff non-portable, Ill-formed CRTL environ value "%s", Ill-formed message in prime_env_iter: |%s|, Illegal binary digit %s, Illegal binary digit %s ignored, Illegal number of bits in vec, Integer overflow in %s number, Invalid %s attribute: %s, Invalid %s attributes: %s, invalid [] range "%s" in regexp, Invalid separator character %s in attribute list, Invalid separator character %s in subroutine attribute list, leaving effective %s failed, Lvalue subs returning %s not implemented yet, Method %s not permitted, Missing %sbrace%s on \N{ }, Missing command in piped open, Missing name in "my sub", No %s specified for -%c, No package name allowed for variable %s in "our", No space allowed after -%c, no UTC offset information; assuming local time is UTC, Octal number 037777777777 non-portable, panic: del_backref, panic: kid popen errno read, panic: magic_killbackrefs, Parentheses missing around "%s" list, Possible unintended interpolation of %s in string, Possible Y2K bug: %s, pragma "attrs" is deprecated, use "sub NAME : ATTRS" instead, Premature end of script headers, Repeat count in pack overflows, Repeat count in unpack overflows, realloc() of freed memory ignored, Reference is already weak, setpgrp can't take arguments, Strange *+?{} on zero-length expression, switching effective %s is not implemented, This Perl can't reset CRTL environ elements (%s), This Perl can't set CRTL environ elements (%s=%s), Too late to run %s block, Unknown open() mode '%s', Unknown process %x sent message to prime_env_iter: %s, Unrecognized escape \\%c passed through, Unterminated attribute parameter in attribute list, Unterminated attribute list, Unterminated attribute parameter in subroutine attribute list, Unterminated subroutine attribute list, Value of CLI symbol "%s" too long, Version number must be a constant number

New tests

Incompatible Changes

Perl Source Incompatibilities

CHECK is a new keyword, Treatment of list slices of undef has changed, Format of \$English::PERL_VERSION is different, Literals of the form 1.2.3 parse differently, Possibly changed pseudo-random number generator, Hashing function for hash keys has changed, undef fails on read only values, Close-on-exec bit may be set on pipe and socket handles, Writing "\$\$1" to mean "\${}\$1" is unsupported, delete(), each(), values() and \(%h), vec(EXPR,OFFSET,BITS) enforces powers-of-two BITS, Text of some diagnostic output has changed, %@ has been removed, Parenthesized not() behaves like a list operator, Semantics of bareword prototype (*) have changed, Semantics of bit operators may have changed on 64-bit platforms, More builtins taint their results

C Source Incompatibilities

PERL_POLLUTE, PERL_IMPLICIT_CONTEXT, PERL_POLLUTE_MALLOC

Compatible C Source API Changes

PATCHLEVEL is now PERL_VERSION

Binary Incompatibilities

Known Problems

Localizing a tied hash element may leak memory

Known test failures

EBCDIC platforms not fully supported

UNICOS/mk CC failures during Configure run

Arrow operator and arrays

Experimental features

Threads, Unicode, 64-bit support, Lvalue subroutines, Weak references, The pseudo-hash data type, The Compiler suite, Internal implementation of file globbing, The DB module, The regular expression code constructs:

Obsolete Diagnostics

Character class syntax [::] is reserved for future extensions, Ill-formed logical name |%s| in prime_env_iter, In string, @%s now must be written as \@%s, Probable precedence problem on %s, regexp too big, Use of "\$\$<digit" to mean "\${}\$<digit" is deprecated

Reporting Bugs
SEE ALSO
HISTORY

perl56delta – what's new for perl v5.6.0

DESCRIPTION

Core Enhancements

- Interpreter cloning, threads, and concurrency
- Lexically scoped warning categories
- Unicode and UTF-8 support
- Support for interpolating named characters
- "our" declarations
- Support for strings represented as a vector of ordinals
- Improved Perl version numbering system
- New syntax for declaring subroutine attributes
- File and directory handles can be autovivified
- `open()` with more than two arguments
- 64-bit support
- Large file support
- Long doubles
- "more bits"
- Enhanced support for `sort()` subroutines
- `sort $coderef @foo` allowed
- File globbing implemented internally
- Support for CHECK blocks
- POSIX character class syntax `[:]` supported
- Better pseudo-random number generator
- Improved `qw//` operator
- Better worst-case behavior of hashes
- `pack()` format 'Z' supported
- `pack()` format modifier '!' supported
- `pack()` and `unpack()` support counted strings
- Comments in `pack()` templates
- Weak references
- Binary numbers supported
- Lvalue subroutines
- Some arrows may be omitted in calls through references
- Boolean assignment operators are legal lvalues
- `exists()` is supported on subroutine names
- `exists()` and `delete()` are supported on array elements
- Pseudo-hashes work better
- Automatic flushing of output buffers
- Better diagnostics on meaningless filehandle operations
- Where possible, buffered data discarded from duped input filehandle
- `eof()` has the same old magic as `<`
- `binmode()` can be used to set `:crlf` and `:raw` modes
- `-T` filetest recognizes UTF-8 encoded files as "text"
- `system()`, backticks and pipe open now reflect `exec()` failure
- Improved diagnostics
- Diagnostics follow STDERR
- More consistent close-on-exec behavior
- `syswrite()` ease-of-use

- Better syntax checks on parenthesized unary operators
- Bit operators support full native integer width
- Improved security features
- More functional bareword prototype (*)
- `require` and `do` may be overridden
- $\X variables may now have names longer than one character
- New variable $\C reflects `-c` switch
- New variable $\V contains Perl version as a string
- Optional Y2K warnings
- Arrays now always interpolate into double-quoted strings
- Modules and Pragmata
 - Modules
 - `attributes`, `B`, `Benchmark`, `ByteLoader`, `constant`, `chardnames`, `Data::Dumper`, `DB`, `DB_File`, `Devel::DProf`, `Devel::Peek`, `Dumpvalue`, `DynaLoader`, `English`, `Env`, `Fcntl`, `File::Compare`, `File::Find`, `File::Glob`, `File::Spec`, `File::Spec::Functions`, `Getopt::Long`, `IO`, `JPL`, `lib`, `Math::BigInt`, `Math::Complex`, `Math::Trig`, `Pod::Parser`, `Pod::InputObjects`, `Pod::Checker`, `podchecker`, `Pod::ParseUtils`, `Pod::Find`, `Pod::Select`, `podselect`, `Pod::Usage`, `pod2usage`, `Pod::Text` and `Pod::Man`, `SDBM_File`, `Sys::Syslog`, `Sys::Hostname`, `Term::ANSIColor`, `Time::Local`, `Win32`, `XSLoader`, `DBM Filters`
 - Pragmata
- Utility Changes
 - `dprofpp`
 - `find2perl`
 - `h2xs`
 - `perlcc`
 - `perldoc`
 - The Perl Debugger
- Improved Documentation
 - `perlapi.pod`, `perlboot.pod`, `perlcompile.pod`, `perldbmfilter.pod`, `perldebug.pod`, `perldebbugs.pod`, `perlfork.pod`, `perlfiler.pod`, `perlhack.pod`, `perlintern.pod`, `perllexwarn.pod`, `perlnumber.pod`, `perlopentut.pod`, `perlrefut.pod`, `perltootc.pod`, `perltodo.pod`, `perlunicode.pod`
- Performance enhancements
 - Simple `sort()` using `{ $a <= $b }` and the like are optimized
 - Optimized assignments to lexical variables
 - Faster subroutine calls
 - `delete()`, `each()`, `values()` and hash iteration are faster
- Installation and Configuration Improvements
 - `-Dusethreads` means something different
 - New Configure flags
 - Threadedness and 64-bitness now more daring
 - Long Doubles
 - `-Dusemorebits`
 - `-Duselargefiles`
 - `installusrbinperl`
 - SOCKS support
 - `-A` flag
 - Enhanced Installation Directories
- Platform specific changes
 - Supported platforms
 - DOS
 - OS390 (OpenEdition MVS)

VMS
 Win32
 Significant bug fixes
 <HANDLE on empty files
 eval '...' improvements
 All compilation errors are true errors
 Implicitly closed filehandles are safer
 Behavior of list slices is more consistent
 (\\$) prototype and \$foo{a}
 goto &sub and AUTOLOAD
 -bareword allowed under use integer
 Failures in DESTROY()
 Locale bugs fixed
 Memory leaks
 Spurious subroutine stubs after failed subroutine calls
 Taint failures under -U
 END blocks and the -c switch
 Potential to leak DATA filehandles
 New or Changed Diagnostics

"%s" variable %s masks earlier declaration in same %s, "my sub" not yet implemented, "our" variable %s redeclared, '!' allowed only after types %s, / cannot take a count, / must be followed by a, A or Z, / must be followed by a*, A* or Z*, / must follow a numeric type, /%s/: Unrecognized escape \\%c passed through, /%s/: Unrecognized escape \\%c in character class passed through, /%s/ should probably be written as "%s", %s() called too early to check prototype, %s argument is not a HASH or ARRAY element, %s argument is not a HASH or ARRAY element or slice, %s argument is not a subroutine name, %s package attribute may clash with future reserved word: %s, (in cleanup) %s, < should be quotes, Attempt to join self, Bad evalled substitution pattern, Bad realloc() ignored, Bareword found in conditional, Binary number 0b11111111111111111111111111111111 non-portable, Bit vector size 32 non-portable, Buffer overflow in prime_env_iter: %s, Can't check filesystem of script "%s", Can't declare class for non-scalar %s in "%s", Can't declare %s in "%s", Can't ignore signal CHLD, forcing to default, Can't modify non-lvalue subroutine call, Can't read CRTL environ, Can't remove %s: %s, skipping file, Can't return %s from lvalue subroutine, Can't weaken a nonreference, Character class [%s:] unknown, Character class syntax [%s] belongs inside character classes, Constant is not %s reference, constant(%s): %s, CORE::%s is not a keyword, defined(@array) is deprecated, defined(%hash) is deprecated, Did not produce a valid header, (Did you mean "local" instead of "our"?), Document contains no data, entering effective %s failed, false [] range "%s" in regexp, Filehandle %s opened only for output, flock() on closed filehandle %s, Global symbol "%s" requires explicit package name, Hexadecimal number 0xffffffff non-portable, Ill-formed CRTL environ value "%s", Ill-formed message in prime_env_iter: [%s], Illegal binary digit %s, Illegal binary digit %s ignored, Illegal number of bits in vec, Integer overflow in %s number, Invalid %s attribute: %s, Invalid %s attributes: %s, invalid [] range "%s" in regexp, Invalid separator character %s in attribute list, Invalid separator character %s in subroutine attribute list, leaving effective %s failed, Lvalue subs returning %s not implemented yet, Method %s not permitted, Missing %sbrace%s on \N{ }, Missing command in piped open, Missing name in "my sub", No %s specified for -%c, No package name allowed for variable %s in "our", No space allowed after -%c, no UTC offset information; assuming local time is UTC, Octal number 03777777777 non-portable, panic: del_backref, panic: kid popen errno read, panic: magic_killbackrefs, Parentheses missing around "%s" list, Possible unintended interpolation of %s in string, Possible Y2K bug: %s, pragma "attrs" is deprecated, use "sub NAME : ATTRS" instead, Premature end of script headers, Repeat count in pack overflows, Repeat count in unpack overflows, realloc() of freed memory ignored, Reference is already weak, setpgrp can't take arguments, Strange *+?{} on zero-length expression, switching effective %s is not implemented, This Perl can't reset CRTL environ elements (%s), This Perl can't set CRTL environ elements (%s=%s), Too late to run %s block, Unknown open() mode '%s', Unknown process %x sent message to prime_env_iter: %s, Unrecognized escape \\%c passed through,

Unterminated attribute parameter in attribute list, Unterminated attribute list, Unterminated attribute parameter in subroutine attribute list, Unterminated subroutine attribute list, Value of CLI symbol "%s" too long, Version number must be a constant number

New tests

Incompatible Changes

Perl Source Incompatibilities

CHECK is a new keyword, Treatment of list slices of undef has changed, Format of `$English::PERL_VERSION` is different, Literals of the form 1.2.3 parse differently, Possibly changed pseudo-random number generator, Hashing function for hash keys has changed, undef fails on read only values, Close-on-exec bit may be set on pipe and socket handles, Writing `$$1` to mean `"${$}1"` is unsupported, `delete()`, `each()`, `values()` and `\(%h)`, `vec(EXPR,OFFSET,BITS)` enforces powers-of-two BITS, Text of some diagnostic output has changed, `%@` has been removed, Parenthesized `not()` behaves like a list operator, Semantics of bareword prototype `(*)` have changed, Semantics of bit operators may have changed on 64-bit platforms, More builtins taint their results

C Source Incompatibilities

`PERL_POLLUTE`, `PERL_IMPLICIT_CONTEXT`, `PERL_POLLUTE_MALLOC`

Compatible C Source API Changes

`PATCHLEVEL` is now `PERL_VERSION`

Binary Incompatibilities

Known Problems

Thread test failures

EBCDIC platforms not supported

In 64-bit HP-UX the `lib/io_multihomed` test may hang

NEXTSTEP 3.3 POSIX test failure

Tru64 (aka Digital UNIX, aka DEC OSF/1) `lib/sdbm` test failure with

`gcc`

UNICOS/mk CC failures during Configure run

Arrow operator and arrays

Experimental features

Threads, Unicode, 64-bit support, Lvalue subroutines, Weak references, The pseudo-hash data type, The Compiler suite, Internal implementation of file globbing, The DB module, The regular expression code constructs:

Obsolete Diagnostics

Character class syntax `[:]` is reserved for future extensions, Ill-formed logical name `[%s]` in `prime_env_iter`, In string, `@@s` now must be written as `\@%s`, Probable precedence problem on `%s`, `regex` too big, Use of `$$<digit>` to mean `"${$}<digit>"` is deprecated

Reporting Bugs

SEE ALSO

HISTORY

perl5005delta – what's new for perl5.005

DESCRIPTION

About the new versioning system

Incompatible Changes

WARNING: This version is not binary compatible with Perl 5.004.

Default installation structure has changed

Perl Source Compatibility

- C Source Compatibility
 - Binary Compatibility
 - Security fixes may affect compatibility
 - Relaxed new mandatory warnings introduced in 5.004
 - Licensing
- Core Changes
 - Threads
 - Compiler
 - Regular Expressions
 - Many new and improved optimizations, Many bug fixes, New regular expression constructs, New operator for precompiled regular expressions, Other improvements, Incompatible changes
 - Improved `malloc()`
 - Quicksort is internally implemented
 - Reliable signals
 - Reliable stack pointers
 - More generous treatment of carriage returns
 - Memory leaks
 - Better support for multiple interpreters
 - Behavior of `local()` on array and hash elements is now well-defined
 - `%!` is transparently tied to the [Errno](#) module
 - Pseudo-hashes are supported
 - `EXPR foreach EXPR` is supported
 - Keywords can be globally overridden
 - `$^E` is meaningful on Win32
 - `foreach (1..1000000)` optimized
 - `Foo::` can be used as implicitly quoted package name
 - `exists $Foo::{Bar::}` tests existence of a package
 - Better locale support
 - Experimental support for 64-bit platforms
 - `prototype()` returns useful results on builtins
 - Extended support for exception handling
 - Re-blessing in `DESTROY()` supported for chaining `DESTROY()` methods
 - All `printf` format conversions are handled internally
 - New `INIT` keyword
 - New `lock` keyword
 - New `qr//` operator
 - `our` is now a reserved word
 - Tied arrays are now fully supported
 - Tied handles support is better
 - 4th argument to `substr`
 - Negative `LENGTH` argument to `splice`
 - Magic lvalues are now more magical
 - `<` now reads in records
- Supported Platforms
 - New Platforms
 - Changes in existing support
- Modules and Pragmata
 - New Modules
 - `B`, `Data::Dumper`, `Dumpvalue`, `Errno`, `File::Spec`, `ExtUtils::Installed`, `ExtUtils::Packlist`, `Fatal`, `IPC::SysV`, `Test`, `Tie::Array`, `Tie::Handle`, `Thread`, `attrs`, `fields`, `re`
 - Changes in existing modules
 - `Benchmark`, `Carp`, `CGI`, `Fcntl`, `Math::Complex`, `Math::Trig`, `POSIX`, `DB_File`, `MakeMaker`, `CPAN`, `Cwd`

Utility Changes

Documentation Changes

New Diagnostics

Ambiguous call resolved as `CORE::%s()`, qualify as such or use `&`, Bad index while coercing array into hash, Bareword `"%s"` refers to nonexistent package, Can't call method `"%s"` on an undefined value, Can't check filesystem of script `"%s"` for nosuid, Can't coerce array into hash, Can't goto subroutine from an eval-string, Can't localize pseudo-hash element, Can't use `%!` because `Errno.pm` is not available, Cannot find an opnumber for `"%s"`, Character class syntax `[. .]` is reserved for future extensions, Character class syntax `[: :]` is reserved for future extensions, Character class syntax `[= =]` is reserved for future extensions, `%s`: Eval-group in insecure regular expression, `%s`: Eval-group not allowed, use `re 'eval'`, `%s`: Eval-group not allowed at run time, Explicit blessing to `"` (assuming package main), Illegal hex digit ignored, No such array field, No such field `"%s"` in variable `%s` of type `%s`, Out of memory during ridiculously large request, Range iterator outside integer range, Recursive inheritance detected while looking for method `'%s' %s`, Reference found where even-sized list expected, Undefined value assigned to `typeglob`, Use of reserved word `"%s"` is deprecated, perl: warning: Setting locale failed

Obsolete Diagnostics

Can't `mktemp()`, Can't write to temp file for `-e: %s`, Cannot open temporary file, regexp too big

Configuration Changes

BUGS

SEE ALSO

HISTORY

perl5004delta – what's new for perl5.004

DESCRIPTION

Supported Environments

Core Changes

List assignment to `%ENV` works

Change to "Can't locate `Foo.pm` in `@INC`" error

Compilation option: Binary compatibility with 5.003

`$PERL5OPT` environment variable

Limitations on `-M`, `-m`, and `-T` options

More precise warnings

Deprecated: Inherited `AUTOLOAD` for non-methods

Previously deprecated `%OVERLOAD` is no longer usable

Subroutine arguments created only when they're modified

Group vector changeable with `$)`

Fixed parsing of `$$<digit`, `&$<digit`, etc.

Fixed localization of `$<digit`, `$&`, etc.

No resetting of `$.` on implicit close

`wantarray` may return `undef`

`eval EXPR` determines value of `EXPR` in scalar context

Changes to tainting checks

No `glob()` or `<*`, No spawning if tainted `$CDPATH`, `$ENV`, `$BASH_ENV`, No spawning if tainted `$TERM` doesn't look like a terminal name

New Opcode module and revised Safe module

Embedding improvements

Internal change: FileHandle class based on `IO::*` classes

Internal change: PerlIO abstraction interface

New and changed syntax

`$coderef-(PARAMS)`

New and changed builtin constants

`__PACKAGE__`

New and changed builtin variables

`$^E`, `$^H`, `$^M`

New and changed builtin functions

`delete` on slices, `flock`, `printf` and `sprintf`, keys as an lvalue, `my()` in Control Structures, `pack()` and `unpack()`, `sysseek()`, use `VERSION`, use Module `VERSION LIST`, `prototype(FUNCTION)`, `srand`, `$_` as Default, `m//gc` does not reset search position on failure, `m//x` ignores whitespace before `?*+{ }`, nested `sub{ }` closures work now, formats work right on changing lexicals

New builtin methods

`isa(CLASS)`, `can(METHOD)`, `VERSION([NEED])`

TIEHANDLE now supported

`TIEHANDLE classname`, `LIST`, `PRINT this`, `LIST`, `PRINTF this`, `LIST`, `READ this` `LIST`, `READLINE this`, `GETC this`, `DESTROY this`

Malloc enhancements

`-DPERL_EMERGENCY_SBRK`, `-DPACK_MALLOC`, `-DTWO_POT_OPTIMIZE`

Miscellaneous efficiency enhancements

Support for More Operating Systems

Win32

Plan 9

QNX

AmigaOS

Pragmata

use `autouse MODULE = qw(sub1 sub2 sub3)`, use `blib`, use `blib 'dir'`, use `constant NAME = VALUE`, use `locale`, use `ops`, use `vmish`

Modules

Required Updates

Installation directories

Module information summary

`Fcntl`

`IO`

`Math::Complex`

`Math::Trig`

`DB_File`

`Net::Ping`

Object-oriented overrides for builtin operators

Utility Changes

`pod2html`

Sends converted HTML to standard output

`xsubpp`

`void XSUBs` now default to returning nothing

C Language API Changes

`gv_fetchmethod` and `perl_call_sv`, `perl_eval_pv`, Extended API for manipulating hashes

Documentation Changes

perldelta, *perlfaq*, *perllocale*, *perltoot*, *perlapi*, *perlmodlib*, *perldebug*, *perlsec*

New Diagnostics

"my" variable %s masks earlier declaration in same scope, %s argument is not a HASH element or slice, Allocation too large: %lx, Allocation too large, Applying %s to %s will act on scalar(%s), Attempt to free nonexistent shared string, Attempt to use reference as lvalue in substr, Bareword "%s" refers to nonexistent package, Can't redefine active sort subroutine %s, Can't use bareword ("%s") as %s ref while "strict refs" in use, Cannot resolve method '%s' overloading '%s' in package '%s', Constant subroutine %s redefined, Constant subroutine %s undefined, Copy method did not return a reference, Died, Exiting pseudo-block via %s, Identifier too long, Illegal character %s (carriage return), Illegal switch in PERL5OPT: %s, Integer overflow in hex number, Integer overflow in octal number, internal error: glob failed, Invalid conversion in %s: "%s", Invalid type in pack: '%s', Invalid type in unpack: '%s', Name "%s::%s" used only once: possible typo, Null picture in formline, Offset outside string, Out of memory!, Out of memory during request for %s, panic: frexp, Possible attempt to put comments in qw() list, Possible attempt to separate words with commas, Scalar value @%s{%s} better written as \$%s{%s}, Stub found while resolving method '%s' overloading '%s' in %s, Too late for "-T" option, untie attempted while %d inner references still exist, Unrecognized character %s, Unsupported function fork, Use of "\$\$<digit" to mean "\${\$}<digit" is deprecated, Value of %s can be "0"; test with defined(), Variable "%s" may be unavailable, Variable "%s" will not stay shared, Warning: something's wrong, Ill-formed logical name |%s| in prime_env_iter, Got an error from DosAllocMem, Malformed PERLLIB_PREFIX, PERL_SH_DIR too long, Process terminated by SIG%s

BUGS

SEE ALSO

HISTORY

perlaix, README.aix – Perl version 5 on IBM Unix (AIX) systems

DESCRIPTION

Compiling Perl 5 on AIX
 OS level
 Building Dynamic Extensions on AIX
 The IBM ANSI C Compiler
 Using GNU's gcc for building perl
 Using Large Files with Perl
 Threaded Perl
 64-bit Perl
 AIX 4.2 and extensions using C++ with statics

AUTHOR

DATE

perlapollo, README.apollo – Perl version 5 on Apollo DomainOS

DESCRIPTION

AUTHOR

perlamiga – Perl under Amiga OS

NOTE

SYNOPSIS

DESCRIPTION

Prerequisites for Compiling Perl on AmigaOS

Unix emulation for AmigaOS: ixemul.library, Version of Amiga OS

- Starting Perl programs under AmigaOS
- Shortcomings of Perl under AmigaOS
- INSTALLATION
 - Accessing documentation
 - Manpages for Perl on AmigaOS
 - Perl HTML Documentation on AmigaOS
 - Perl GNU Info Files on AmigaOS
 - Perl LaTeX Documentation on AmigaOS
- BUILDING PERL ON AMIGAOS
 - Build Prerequisites for Perl on AmigaOS
 - Getting the Perl Source for AmigaOS
 - Making Perl on AmigaOS
 - Testing Perl on AmigaOS
 - Installing the built Perl on AmigaOS
- PERL 5.8.0 BROKEN IN AMIGAOS
- AUTHORS
- SEE ALSO

perlbeos, README.beos – Perl version 5 on BeOS

- DESCRIPTION
 - General Issues with Perl on BeOS
 - BeOS Release-specific Notes
 - R4 x86, R4 PPC
 - Contact Information
 - Update 2002-05-30

perlbs2000, README.BS2000 – building and installing Perl for BS2000.

- SYNOPSIS
- DESCRIPTION
 - gzip on BS2000
 - bison on BS2000
 - Unpacking Perl Distribution on BS2000
 - Compiling Perl on BS2000
 - Testing Perl on BS2000
 - Installing Perl on BS2000
 - Using Perl in the Posix-Shell of BS2000
 - Using Perl in "native" BS2000
 - Floating point anomalies on BS2000
- AUTHORS
- SEE ALSO
 - Mailing list
- HISTORY
 - Name
 - Description
 - Build
 - Tools & SDK
 - Make
 - Acknowledgements
 - Author

perlcygwin, README.cygwin – Perl for Cygwin

- SYNOPSIS

PREREQUISITES FOR COMPILING PERL ON CYGWIN

Cygwin = GNU+Cygnus+Windows (Don't leave UNIX without it)

Cygwin Configuration

`PATH`, `nroff`, Permissions

CONFIGURE PERL ON CYGWIN

Stripping Perl Binaries on Cygwin

Optional Libraries for Perl on Cygwin

`-lcrypt`, `-lgdbm` (use `GDBM_File`), `-ldb` (use `DB_File`), `-lcygipc` (use `IPC::SysV`), `-lutil`

Configure-time Options for Perl on Cygwin

`-Uusedl`, `-Uusemymalloc`, `-Uuseperlio`, `-Dusemultiplicity`, `-Duse64bitint`,
`-Duselongsdouble`, `-Dusetreads`, `-Duselargefiles`, `-Dmksymlinks`

Suspicious Warnings on Cygwin

`dlsym()`, `Win9x` and `d_eofnblk`, Compiler/Preprocessor defines

MAKE ON CYGWIN

Warnings on Cygwin

`ld2` on Cygwin

TEST ON CYGWIN

File Permissions on Cygwin

`NDBM_File` does not work on FAT filesystems

Script Portability on Cygwin

Pathnames, Text/Binary, `.exe`, `chown()`, Miscellaneous

INSTALL PERL ON CYGWIN**MANIFEST ON CYGWIN**

Documentation, Build, Configure, Make, Install, Tests, Compiled Perl Source, Compiled Module
Source, Perl Modules/Scripts

BUGS ON CYGWIN**AUTHORS****HISTORY****perlDgux – Perl under DG/UX.****SYNOPSIS****DESCRIPTION****BUILDING PERL ON DG/UX**

Non-threaded Perl on DG/UX

Threaded Perl on DG/UX

Testing Perl on DG/UX

Installing the built perl on DG/UX

AUTHOR**SEE ALSO****perlDdos – Perl under DOS, W31, W95.****SYNOPSIS****DESCRIPTION**

Prerequisites for Compiling Perl on DOS

DJGPP, Pthreads

Shortcomings of Perl under DOS

Building Perl on DOS

- Testing Perl on DOS
- Installation of Perl on DOS
- BUILDING AND INSTALLING MODULES ON DOS
 - Building Prerequisites for Perl on DOS
 - Unpacking CPAN Modules on DOS
 - Building Non-XS Modules on DOS
 - Building XS Modules on DOS
- AUTHOR
- SEE ALSO

perlepoc, README.epoc – Perl for EPOC

- SYNOPSIS
- INTRODUCTION
- INSTALLING PERL ON EPOC
- STARTING PERL ON EPOC
 - Editors on Epoc
 - Features of Perl on Epoc
 - Restrictions of Perl on Epoc
 - Compiling Perl 5 on the EPOC cross compiling environment
- SUPPORT STATUS OF PERL ON EPOC
- AUTHOR
- LAST UPDATE

perlfreebsd, README.freebsd – Perl version 5 on FreeBSD systems

- DESCRIPTION
 - FreeBSD core dumps from readdir_r with ithreads
 - \$^x doesn't always contain a full path in FreeBSD
 - Perl will no longer be part of "base FreeBSD"
- AUTHOR

perlhpux, README.hpux – Perl version 5 on Hewlett-Packard Unix

(HP-UX) systems

- DESCRIPTION
 - Using perl as shipped with HP-UX
 - Compiling Perl 5 on HP-UX
 - PA-RISC
 - PA-RISC 1.0
 - PA-RISC 1.1
 - PA-RISC 2.0
 - Itanium
 - Portability Between PA-RISC Versions
 - Itanium Processor Family and HP-UX
 - Building Dynamic Extensions on HP-UX
 - The HP ANSI C Compiler
 - The GNU C Compiler
 - Using Large Files with Perl on HP-UX
 - Threaded Perl on HP-UX
 - 64-bit Perl on HP-UX
 - Oracle on HP-UX
 - GDBM and Threads on HP-UX
 - NFS filesystems and utime(2) on HP-UX
 - perl -P and // and HP-UX
 - HP-UX Kernel Parameters (maxdsiz) for Compiling Perl

nss_delete core dump from op/pwent or op/grent
AUTHOR
DATE

perlhurd, README.hurd – Perl version 5 on Hurd

DESCRIPTION
Known Problems with Perl on Hurd
AUTHOR

perlirix, README.irix – Perl version 5 on Irix systems

DESCRIPTION
Building 32-bit Perl in Irix
Building 64-bit Perl in Irix
About Compiler Versions of Irix
Linker Problems in Irix
Malloc in Irix
Building with threads in Irix
AUTHOR

perlmachten, README.machten – Perl version 5 on Power MachTen

systems
DESCRIPTION
Compiling Perl 5 on MachTen
Failures during `make test` on MachTen
`op/lexassign.t`, `pragma/warnings.t`
Building external modules on MachTen
AUTHOR
DATE

perlmacos, README.macos – Perl under Mac OS (Classic)

SYNOPSIS
DESCRIPTION
AUTHOR
DATE

perlmint, README.mint – Perl version 5 on Atari MiNT

DESCRIPTION
Known problems with Perl on MiNT
AUTHOR

perlmpaix, README.mpeix – Perl/iX for HP e3000 MPE

SYNOPSIS
NOTE
Binary distribution from HP
What's New in Perl for MPE/iX
Welcome to Perl/iX
System Requirements for Perl/iX
How to Obtain Perl/iX
Perl/iX Distribution Contents Highlights
README, INSTALL, LIBSHP3K, PERL, .cpan/, lib/, man/, public_html/feedback.cgi,
src/perl-5.6.0-mpe

How to Compile Perl/iX

4, 6

Getting Started with Perl/iX

MPE/iX Implementation Considerations

Known Perl/iX Bugs Under Investigation

Perl/iX To-Do List

Perl/iX Change History

AUTHOR

Name

Description

Build

Tools & SDK

Setup

SetNWBld.bat, Buildtype.bat

Make

Interpreter

Extensions

Install

Build new extensions

Acknowledgements

Authors

Date

perlos2 – Perl under OS/2, DOS, Win0.3*, Win0.95 and WinNT.

SYNOPSIS

DESCRIPTION

Target

Other OSes

Prerequisites

EMX, RSX, HPFS, pdksh

Starting Perl programs under OS/2 (and DOS and...)

Starting OS/2 (and DOS) programs under Perl

Frequently asked questions

"It does not work"

I cannot run external programs

I cannot embed perl into my program, or use **perl.dll** from my program.

Is your program EMX-compiled with `-Zmt -Zcrt.dll?`, Did you use [ExtUtils::Embed](#)?

`` `` and `pipe-open` do not work under DOS.

Cannot start `find.exe` "pattern" file

INSTALLATION

Automatic binary installation

`PERL_BADLANG`, `PERL_BADFREE`, *Config.pm*

Manual binary installation

Perl VIO and PM executables (dynamically linked), Perl_VIO executable (statically linked), Executables for Perl utilities, Main Perl library, Additional Perl modules, Tools to compile Perl modules, Manpages for Perl and utilities, Manpages for Perl modules, Source for Perl documentation, Perl manual in *.INF* format, Pdksh

Warning

Accessing documentation

OS/2 **.INF** file

Plain text

Manpages

HTML

GNU info files

PDF files

LaTeX docs

BUILD

The short story

Prerequisites

Getting perl source

Application of the patches

Hand-editing

Making

Testing

A lot of bad free, Process terminated by SIGTERM/SIGINT, *op/fs.t*, *op/stat.t*

Installing the built perl

a.out-style build

Build FAQ

Some / became \ in pdksh.

'errno' - unresolved external

Problems with tr or sed

Some problem (forget which ;-)

Library ... not found

Segfault in make

op/sprintf test failure

Specific (mis)features of OS/2 port

setpriority, getpriority

system()

extproc on the first line

Additional modules:

Prebuilt methods:

```
File::Copy::syscopy, DynaLoader::mod2fname, Cwd::current_drive(),
Cwd::sys_chdir(name), Cwd::change_drive(name),
Cwd::sys_is_absolute(name), Cwd::sys_is_rooted(name),
Cwd::sys_is_relative(name), Cwd::sys_cwd(name),
Cwd::sys_abspath(name, dir), Cwd::extLibpath([type]),
Cwd::extLibpath_set( path [, type ] ),
OS2::Error(do_harderror, do_exception), OS2::Errors2Drive(drive),
OS2::SysInfo(), OS2::BootDrive(), OS2::MorphPM(serve),
OS2::UnMorphPM(serve), OS2::Serve_Messages(force),
OS2::Process_Messages(force [, cnt]), OS2::_control87(new, mask),
OS2::get_control87(),
OS2::set_control87_em(new=MCW_EM, mask=MCW_EM), OS2::DLLname([how
[, \&xsub]])
```

Prebuilt variables:

```
$OS2::emx_rev, $OS2::emx_env, $OS2::os_ver, $OS2::is_aout,
$OS2::can_fork, $OS2::nsyserror
```

Misfeatures

Modifications

`popen`, `tmpnam`, `tmpfile`, `ctermid`, `stat`, `mkdir`, `rmdir`, `flock`

Identifying DLLs

Centralized management of resources

`HAB`, `HMQ`, Treating errors reported by OS/2 API, `CheckOSError (expr)`,
`CheckWinError (expr)`, `SaveWinError (expr)`,
`SaveCroakWinError (expr, die, name1, name2)`, `WinError_2_Perl_rc`,
`FillWinError`, `FilloSError (rc)`, Loading DLLs and ordinals in DLLs

Perl flavors

perl.exe

perl_.exe

perl___.exe

perl____.exe

Why strange names?

Why dynamic linking?

Why chimera build?

ENVIRONMENT

`PERLLIB_PREFIX`

`PERL_BADLANG`

`PERL_BADFREE`

`PERL_SH_DIR`

`USE_PERL_FLOCK`

`TMP` or `TEMP`

Evolution

Text-mode filehandles

Priorities

DLL name mangling: pre 5.6.2

DLL name mangling: 5.6.2 and beyond

Global DLLs, specific DLLs, `BEGINLIBPATH` and `ENDLIBPATH`, . from `LIBPATH`

DLL forwarder generation

Threading

Calls to external programs

Memory allocation

Threads

`COND_WAIT`, *os2.c*

BUGS

AUTHOR

SEE ALSO

perlos390, README.os390 – building and installing Perl for OS/390

and z/OS

SYNOPSIS

DESCRIPTION

Tools

Unpacking Perl distribution on OS/390

Setup and utilities for Perl on OS/390

Configure Perl on OS/390

Build, Test, Install Perl on OS/390

- Build Anomalies with Perl on OS/390
- Testing Anomalies with Perl on OS/390
- Installation Anomalies with Perl on OS/390
- Usage Hints for Perl on OS/390
- Floating Point Anomalies with Perl on OS/390
- Modules and Extensions for Perl on OS/390

AUTHORS

SEE ALSO

- Mailing list for Perl on OS/390

HISTORY

perlqnx, README.qnx – Perl version 5 on QNX

DESCRIPTION

- Required Software for Compiling Perl on QNX4
 - /bin/sh, ar, nm, cpp, make
- Outstanding Issues with Perl on QNX4
- QNX auxiliary files
 - qnx/ar, qnx/cpp
- Outstanding issues with perl under QNX6

AUTHOR

perlplan9 – Plan 9–specific documentation for Perl

DESCRIPTION

- Invoking Perl
- What's in Plan 9 Perl
- What's not in Plan 9 Perl
- Perl5 Functions not currently supported in Plan 9 Perl
- Signals in Plan 9 Perl

COMPILING AND INSTALLING PERL ON PLAN 9

- Installing Perl Documentation on Plan 9

BUGS

Revision date

AUTHOR

perlsolaris, README.solaris – Perl version 5 on Solaris systems

DESCRIPTION

- Solaris Version Numbers.

RESOURCES

- Solaris FAQ, Precompiled Binaries, Solaris Documentation

SETTING UP

- File Extraction Problems on Solaris.
- Compiler and Related Tools on Solaris.
- Environment for Compiling Perl on Solaris

RUN CONFIGURE.

- 64–bit Issues with Perl on Solaris.
- Threads in Perl on Solaris.
- Malloc Issues with Perl on Solaris.

MAKE PROBLEMS.

- Dynamic Loading Problems With GNU as and GNU ld, ld.so.1: ./perl: fatal: relocation error: dlopen: stub interception failed, #error "No DATAMODEL_NATIVE specified", sh: ar: not found

MAKE TEST
 op/stat.t test 4 in Solaris
 nss_delete core dump from op/pwent or op/grent
PREBUILT BINARIES OF PERL FOR SOLARIS.
RUNTIME ISSUES FOR PERL ON SOLARIS.
 Limits on Numbers of Open Files on Solaris.
SOLARIS-SPECIFIC MODULES.
SOLARIS-SPECIFIC PROBLEMS WITH MODULES.
 Proc::ProcessTable on Solaris
 BSD::Resource on Solaris
 Net::SSLeay on Solaris
AUTHOR
LAST MODIFIED

perltru64, README.tru64 – Perl version 5 on Tru64 (formerly known as

Digital UNIX formerly known as DEC OSF/1) systems

DESCRIPTION
 Compiling Perl 5 on Tru64
 Using Large Files with Perl on Tru64
 Threaded Perl on Tru64
 Long Doubles on Tru64
 64-bit Perl on Tru64
 Warnings about floating-point overflow when compiling Perl on Tru64
Testing Perl on Tru64
ext/ODBM_File/odbm Test Failing With Static Builds
Perl Fails Because Of Unresolved Symbol socketmark
AUTHOR

perluts – Perl under UTS

SYNOPSIS
DESCRIPTION
BUILDING PERL ON UTS
Installing the built perl on UTS
AUTHOR

perlvmesa, README.vmesa – building and installing Perl for VM/ESA.

SYNOPSIS
DESCRIPTION
 Unpacking Perl Distribution on VM/ESA
 Setup Perl and utilities on VM/ESA
 Configure Perl on VM/ESA
 Testing Anomalies of Perl on VM/ESA
 Usage Hints for Perl on VM/ESA
AUTHORS
SEE ALSO
 Mailing list for Perl on VM/ESA

perlvms – VMS-specific documentation for Perl

DESCRIPTION
Installation
Organization of Perl Images
 Core Images

- Perl Extensions
 - Installing static extensions
 - Installing dynamic extensions
- File specifications
 - Syntax
 - Wildcard expansion
 - Pipes
- PERL5LIB and PERLLIB
- Command line
 - I/O redirection and backgrounding
 - Command line switches
 - i, -S, -u
- Perl functions
 - File tests, backticks, binmode FILEHANDLE, crypt PLAINTEXT, USER, dump, exec LIST, fork, getpwent, getpwnam, getpwuid, gmtime, kill, qx//, select (system call), stat EXPR, system LIST, time, times, unlink LIST, utime LIST, waitpid PID, FLAGS
- Perl variables
 - %ENV, CRTL_ENV, CLISYM_[LOCAL], Any other string, \$!, \$^E, \$?, \$|
- Standard modules with VMS-specific differences
 - SDBM_File
- Revision date
- AUTHOR

perlvos, README.vos – Perl for Stratus VOS

- SYNOPSIS
 - Multiple methods to build perl for VOS
 - Stratus POSIX Support
- INSTALLING PERL IN VOS
 - Compiling Perl 5 on VOS
 - Installing Perl 5 on VOS
- USING PERL IN VOS
 - Unimplemented Features of Perl on VOS
 - Restrictions of Perl on VOS
 - Handling of underflow and overflow
- TEST STATUS
- SUPPORT STATUS
- AUTHOR
- LAST UPDATE

perlwin32 – Perl under Windows

- SYNOPSIS
- DESCRIPTION
 - Setting Up Perl on Win32
 - Make, Command Shell, Borland C++, Microsoft Visual C++, Microsoft Platform SDK 64-bit Compiler, Mingw32 with GCC
 - Building
 - Testing Perl on Win32
 - Installation of Perl on Win32
 - Usage Hints for Perl on Win32
 - Environment Variables, File Globbing, Using perl from the command line, Building Extensions, Command-line Wildcard Expansion, Win32 Specific Extensions, Notes on 64-bit Windows

Running Perl Scripts

Miscellaneous Things

BUGS AND CAVEATS

AUTHORS

Gary Ng <71564.1743@CompuServe.COM>, Gurusamy Sarathy <gsar@activestate.com>, Nick
Ing-Simmons <nick@ing-simmons.net>

SEE ALSO

HISTORY

PRAGMA DOCUMENTATION

attrs – set/get attributes of a subroutine (deprecated)

SYNOPSIS

DESCRIPTION

method, locked

re – Perl pragma to alter regular expression behaviour

SYNOPSIS

DESCRIPTION

threadshared::shared, threads::shared – Perl extension for sharing

data structures between threads

SYNOPSIS

DESCRIPTION

EXPORT

FUNCTIONS

share VARIABLE, lock VARIABLE, cond_wait VARIABLE, cond_signal VARIABLE,
cond_broadcast VARIABLE

NOTES

BUGS

AUTHOR

SEE ALSO

threads – Perl extension allowing use of interpreter based threads

from perl

SYNOPSIS

DESCRIPTION

\$thread = threads-create(function, LIST), \$thread-join, \$thread-detach, threads-self,
\$thread-tid, threads-object(tid), threads-yield();, threads-list();, async BLOCK;

WARNINGS

A thread exited while %d other threads were still running

TODO

BUGS

Parent-Child threads, Returning objects, Creating threads inside BEGIN blocks,
PERL_OLD_SIGNALS are not threadsafe, will not be

AUTHOR and COPYRIGHT

SEE ALSO

attributes – get/set subroutine or variable attributes

SYNOPSIS

DESCRIPTION

Built-in Attributes

locked, method, lvalue

Available Subroutines

get, reftype

Package-specific Attribute Handling

FETCH_type_ATTRIBUTES, MODIFY_type_ATTRIBUTES

Syntax of Attribute Lists

EXPORTS

Default exports

Available exports

Export tags defined

EXAMPLES

SEE ALSO

attrs – set/get attributes of a subroutine (deprecated)

SYNOPSIS

DESCRIPTION

method, locked

autouse – postpone load of modules until a function is used

SYNOPSIS

DESCRIPTION

WARNING

AUTHOR

SEE ALSO

base – Establish IS-A relationship with base class at compile time

SYNOPSIS

DESCRIPTION

HISTORY

SEE ALSO

bigint – Transparent big integer support for Perl

SYNOPSIS

DESCRIPTION

OPTIONS

a or accuracy, p or precision, t or trace, l or lib, v or version

MATH LIBRARY

INTERNAL FORMAT

SIGN

METHODS

MODULES USED

EXAMPLES

LICENSE

SEE ALSO

AUTHORS

bignum – Transparent BigNumber support for Perl

SYNOPSIS

DESCRIPTION

OPTIONS

a or accuracy, p or precision, t or trace, l or lib, v or version

MATH LIBRARY

INTERNAL FORMAT

SIGN

METHODS

MODULES USED

EXAMPLES

LICENSE

SEE ALSO

AUTHORS

bigrat – Transparent BigNumber/BigRational support for Perl

SYNOPSIS

DESCRIPTION

MODULES USED

MATH LIBRARY

SIGN

METHODS

EXAMPLES

```
perl -Mbigrat -le 'print sqrt(33)'  
perl -Mbigrat -le 'print 2*255'  
perl -Mbigrat -le 'print 4.5+2*255'  
perl -Mbigrat -le 'print 3/7 + 5/7 + 8/3'  
perl -Mbigrat -le 'print 12->is_odd()';
```

LICENSE

SEE ALSO

AUTHORS

blib – Use MakeMaker's uninstalled version of a package

SYNOPSIS

DESCRIPTION

BUGS

AUTHOR

bytes – Perl pragma to force byte semantics rather than character

semantics

SYNOPSIS

DESCRIPTION

SEE ALSO

charnames – define character names for `\N{named}` string literal

escapes

SYNOPSIS

DESCRIPTION

CUSTOM TRANSLATORS

charnames::viacode(code)

chardnames::vianame(name)
ALIASES
ILLEGAL CHARACTERS
BUGS

constant – Perl pragma to declare constants

SYNOPSIS
DESCRIPTION
NOTES
 List constants
 Defining multiple constants at once
 Magic constants
TECHNICAL NOTES
BUGS
AUTHOR
COPYRIGHT

diagnostics – Perl compiler pragma to force verbose warning

diagnostics
SYNOPSIS
DESCRIPTION
 The `diagnostics` Pragma
 The *splain* Program
EXAMPLES
INTERNALS
BUGS
AUTHOR

encoding – allows you to write your script in non-ascii or non-utf8

SYNOPSIS
ABSTRACT
USAGE
 use encoding [*ENCNAME*] ;, use encoding *ENCNAME* [*STDIN => ENCNAME_IN ...*] ;, no encoding;

CAVEATS
 NOT SCOPED
 DO NOT MIX MULTIPLE ENCODINGS
Non-ASCII Identifiers and Filter option
 use encoding *ENCNAME* Filter=>1;

EXAMPLE – Greekperl
KNOWN PROBLEMS
SEE ALSO

fields – compile-time class fields

SYNOPSIS
DESCRIPTION
 new, phash

SEE ALSO

filetest – Perl pragma to control the filetest permission operators

SYNOPSIS

DESCRIPTION
subpragma access

if – use a Perl module if a condition holds

SYNOPSIS
DESCRIPTION
BUGS
AUTHOR

integer – Perl pragma to use integer arithmetic instead of floating

point

SYNOPSIS
DESCRIPTION

less – perl pragma to request less of something from the compiler

SYNOPSIS
DESCRIPTION

lib – manipulate @INC at compile time

SYNOPSIS
DESCRIPTION
Adding directories to @INC
Deleting directories from @INC
Restoring original @INC

CAVEATS
NOTES
SEE ALSO
AUTHOR

locale – Perl pragma to use and avoid POSIX locales for built-in

operations

SYNOPSIS
DESCRIPTION

open – perl pragma to set default PerlIO layers for input and output

SYNOPSIS
DESCRIPTION
NONPERLIO FUNCTIONALITY
IMPLEMENTATION DETAILS
SEE ALSO

ops – Perl pragma to restrict unsafe operations when compiling

SYNOPSIS
DESCRIPTION
SEE ALSO

overload – Package for overloading perl operations

SYNOPSIS
DESCRIPTION
Declaration of overloaded functions
Calling Conventions for Binary Operations
FALSE, TRUE, undef

Calling Conventions for Unary Operations

Calling Conventions for Mutators

++ and --, x= and other assignment versions

Overloadable Operations

Arithmetic operations, Comparison operations, Bit operations, Increment and decrement, Transcendental functions, Boolean, string and numeric conversion, Iteration, Dereferencing, Special

Inheritance and overloading

Strings as values of use overload directive, Overloading of an operation is inherited by derived classes

SPECIAL SYMBOLS FOR use overload

Last Resort

Fallback

undef, TRUE, defined, but FALSE

Copy Constructor

Example

MAGIC AUTOGENERATION

Assignment forms of arithmetic operations, Conversion operations, Increment and decrement, abs (\$a) , Unary minus, Negation, Concatenation, Comparison operations, Iterator, Dereferencing, Copy operator

Losing overloading

Run-time Overloading

Public functions

overload::StrVal(arg), overload::Overloaded(arg), overload::Method(obj,op)

Overloading constants

integer, float, binary, q, qr

IMPLEMENTATION

Metaphor clash

Cookbook

Two-face scalars

Two-face references

Symbolic calculator

Really symbolic calculator

AUTHOR

DIAGNOSTICS

Odd number of arguments for overload::constant, '%s' is not an overloadable type, '%s' is not a code reference

BUGS

re – Perl pragma to alter regular expression behaviour

SYNOPSIS

DESCRIPTION

sigtrap – Perl pragma to enable simple signal handling

SYNOPSIS

DESCRIPTION

OPTIONS

SIGNAL HANDLERS

stack-trace, **die**, **handler** *your-handler*

SIGNAL LISTS

normal-signals, **error-signals**, **old-interface-signals**

OTHER

untrapped, **any**, *signal*, *number*

EXAMPLES

sort – perl pragma to control `sort()` behaviour

SYNOPSIS

DESCRIPTION

CAVEATS

strict – Perl pragma to restrict unsafe constructs

SYNOPSIS

DESCRIPTION

`strict refs`, `strict vars`, `strict subs`

subs – Perl pragma to predeclare sub names

SYNOPSIS

DESCRIPTION

threads – Perl extension allowing use of interpreter based threads

from perl

SYNOPSIS

DESCRIPTION

`$thread = threads-create(function, LIST)`, `$thread-join`, `$thread-detach`, `threads-self`,
`$thread-tid`, `threads-object(tid)`, `threads-yield()`, `threads-list()`, `async BLOCK`;

WARNINGS

A thread exited while %d other threads were still running

TODO

BUGS

Parent-Child threads, Returning objects, Creating threads inside BEGIN blocks,
PERL_OLD_SIGNALS are not threadsafe, will not be

AUTHOR and COPYRIGHT

SEE ALSO

threadshared, threads::shared – Perl extension for sharing data

structures between threads

SYNOPSIS

DESCRIPTION

EXPORT

FUNCTIONS

`share VARIABLE`, `lock VARIABLE`, `cond_wait VARIABLE`, `cond_signal VARIABLE`,
`cond_broadcast VARIABLE`

NOTES

BUGS
AUTHOR
SEE ALSO

utf8 – Perl pragma to enable/disable UTF-8 (or UTF-EBCDIC) in source

code

SYNOPSIS
DESCRIPTION

Utility functions

```
$num_octets = utf8::upgrade($string);, utf8::downgrade($string[, FAIL_OK]),
utf8::encode($string), $flag=utf8::decode($string), $flag=utf8::valid(String)
```

BUGS
SEE ALSO

vars – Perl pragma to predeclare global variable names (obsolete)

SYNOPSIS
DESCRIPTION

vmsish – Perl pragma to control VMS-specific language features

SYNOPSIS
DESCRIPTION

```
vmsish status,vmsish exit,vmsish time,vmsish hushed
```

warnings – Perl pragma to control optional warnings

SYNOPSIS
DESCRIPTION

```
use warnings::register, warnings::enabled(), warnings::enabled($category),
warnings::enabled($object), warnings::warn($message), warnings::warn($category,
$message), warnings::warn($object, $message), warnings::warnif($message),
warnings::warnif($category, $message), warnings::warnif($object, $message)
```

warnings::register – warnings import function

SYNOPSIS
DESCRIPTION

MODULE DOCUMENTATION

AnyDBM_File – provide framework for multiple DBMs

SYNOPSIS
DESCRIPTION
DBM Comparisons
[0], [1], [2], [3]
SEE ALSO

Attribute::Handlers – Simpler definition of attribute handlers

VERSION
SYNOPSIS
DESCRIPTION
[0], [1], [2], [3], [4], [5]
Typed lexicals
Type-specific attribute handlers

Non-interpretive attribute handlers

Phase-specific attribute handlers

Attributes as tie interfaces

EXAMPLES

DIAGNOSTICS

Bad attribute type: ATTR(%s), Attribute handler %s doesn't handle %s attributes, Declaration of %s attribute in package %s may clash with future reserved word, Can't have two ATTR specifiers on one subroutine, Can't autotie a %s, Internal error: %s symbol went missing, Won't be able to apply END handler

AUTHOR

BUGS

COPYRIGHT

AutoLoader – load subroutines only on demand

SYNOPSIS

DESCRIPTION

Subroutine Stubs

Using **AutoLoader**'s AUTOLOAD Subroutine

Overriding **AutoLoader**'s AUTOLOAD Subroutine

Package Lexicals

Not Using AutoLoader

AutoLoader vs. **SelfLoader**

CAVEATS

SEE ALSO

AutoSplit – split a package for autoloading

SYNOPSIS

DESCRIPTION

\$keep, \$check, \$modtime

Multiple packages

DIAGNOSTICS

B – The Perl Compiler

SYNOPSIS

DESCRIPTION

OVERVIEW OF CLASSES

SV-RELATED CLASSES

B::SV METHODS

REFCNT, FLAGS

B::IV METHODS

IV, IVX, UVX, int_value, needs64bits, packiv

B::NV METHODS

NV, NVX

B::RV METHODS

RV

B::PV METHODS

PV, RV, PVX

B::PVMG METHODS

MAGIC, SvSTASH

B::MAGIC METHODS

MOREMAGIC, precomp, PRIVATE, TYPE, FLAGS, OBJ, PTR, REGEX

B::PVLV METHODS

TARGOFF, TARGLEN, TYPE, TARG

B::BM METHODS

USEFUL, PREVIOUS, RARE, TABLE

B::GV METHODS

is_empty, NAME, SAFENAME, STASH, SV, IO, FORM, AV, HV, EGV, CV, CVGEN, LINE, FILE, FILEGV, GvREFCNT, FLAGS

B::IO METHODS

LINES, PAGE, PAGE_LEN, LINES_LEFT, TOP_NAME, TOP_GV, FMT_NAME, FMT_GV, BOTTOM_NAME, BOTTOM_GV, SUBPROCESS, IoTYPE, IoFLAGS, IsSTD

B::AV METHODS

FILL, MAX, OFF, ARRAY, AvFLAGS

B::CV METHODS

STASH, START, ROOT, GV, FILE, DEPTH, PADLIST, OUTSIDE, XSUB, XSUBANY, CvFLAGS, const_sv

B::HV METHODS

FILL, MAX, KEYS, RITER, NAME, PMROOT, ARRAY

OP-RELATED CLASSES**B::OP METHODS**

next, sibling, name, ppaddr, desc, targ, type, seq, flags, private

B::UNOP METHOD

first

B::BINOP METHOD

last

B::LOGOP METHOD

other

B::LISTOP METHOD

children

B::PMOP METHODS

pmreplroot, pmreplstart, pmnext, pmregexp, pmflags, pmdynflags, pmpermflags, precomp, pmoffet

B::SVOP METHOD

sv, gv

B::PADOP METHOD

padix

B::PVOP METHOD

pv

B::LOOP METHODS

redoop, nextop, lastop

B::COP METHODS

label, stash, file, cop_seq, arybase, line

FUNCTIONS EXPORTED BY B

main_cv, init_av, begin_av, end_av, main_root, main_start, comppadlist, regex_padav, sv_undef, sv_yes, sv_no, amagic_generation, walkoptree(OP, METHOD), walkoptree_debug(DEBUG), walksymtable(SYMREF, METHOD, RECURSE, PREFIX), svref_2object(SV), ppname(OPNUM), hash(STR), cast_I32(I), minus_c, cstring(STR), perlstring(STR), class(OBJ), threadsv_names

AUTHOR**B::Asmdata – Autogenerated data about Perl ops, used to generate**

bytecode

SYNOPSIS**DESCRIPTION**

%insn_data, @insn_name, @optype, @specialsv_name

AUTHOR**B::Assembler – Assemble Perl bytecode****SYNOPSIS****DESCRIPTION****AUTHORS****B::Bblock – Walk basic blocks****SYNOPSIS****DESCRIPTION**

Functions

find_leaders**AUTHOR****B::Bytecode – Perl compiler's bytecode backend****SYNOPSIS****DESCRIPTION****OPTIONS**

–ofilename, –afilename, —, –f, –fcompress–nullops, –fomit–sequence–numbers,
–fbypass–nullops, –On, –D, –Do, –Db, –Da, –DC, –S, –upackage

EXAMPLES**BUGS****AUTHORS****B::C – Perl compiler's C backend****SYNOPSIS****DESCRIPTION****OPTIONS**

–ofilename, –v, —, –uPackname, –D, –Do, –Dc, –DA, –DC, –DM, –f, –fcog, –fsave–data,
–fppaddr, –fwarn–sv, –fuse–script–name, –fsave–sig–hash, –On, –O0, –O1, –O2, –llimit

EXAMPLES

BUGS
AUTHOR

B::CC – Perl compiler's optimized C translation backend

SYNOPSIS
DESCRIPTION
OPTIONS

-ofilename, -v, —, -uPackname, -mModulename, -D, -Dr, -DO, -Ds, -Dp, -Dq, -Dl, -Dt, -f, -ffreetmps-each-bblock, -ffreetmps-each-loop, -fomit-taint, -On

EXAMPLES

BUGS

DIFFERENCES

Loops
Context of ".."
Arithmetic
Deprecated features

AUTHOR

B::Concise – Walk Perl syntax tree, printing concise info about ops

SYNOPSIS
DESCRIPTION
EXAMPLE
OPTIONS

-basic, -exec, -tree, -compact, -loose, -vt, -ascii, -main, -basen, -bigendian, -littleendian, -concise, -terse, -linenoise, -debug, -env

FORMATTING SPECIFICATIONS

(*x(exec_text;basic_text)x*), ((text)**), (**(text1;text2)**), (*?(text1#varText2)?*), *#var*, *#varN*, *~*, *#addr*, *#arg*, *#class*, *#classsym*, *#coplabel*, *#exname*, *#extarg*, *#firstaddr*, *#flags*, *#flagval*, *#hyphseq*, *#label*, *#lastaddr*, *#name*, *#NAME*, *#next*, *#nextaddr*, *#noise*, *#private*, *#privval*, *#seq*, *#seqnum*, *#sibaddr*, *#svaddr*, *#svclass*, *#svval*, *#targ*, *#targarg*, *#targarglife*, *#typenum***

ABBREVIATIONS

OP flags abbreviations
OP class abbreviations

Using B::Concise outside of the O framework

AUTHOR

B::Debug – Walk Perl syntax tree, printing debug info about ops

SYNOPSIS
DESCRIPTION
AUTHOR

B::Deparse – Perl compiler backend to produce perl code

SYNOPSIS
DESCRIPTION
OPTIONS

-l, -p, -P, -q, -fFILE, -sLETTERS, C, iNUMBER, T, vSTRING., -xLEVEL

USING B::Deparse AS A MODULE

Synopsis
Description
new

ambient_pragmas
strict, \$ [, bytes, utf8, integer, re, warnings, hint_bits, warning_bits
coderef2text
BUGS
AUTHOR

B::Disassembler – Disassemble Perl bytecode

SYNOPSIS
DESCRIPTION
AUTHOR

B::Lint – Perl lint

SYNOPSIS
DESCRIPTION
OPTIONS AND LINT CHECKS
context, implicit-read and implicit-write, dollar-underscore, private-names, undefined-subs,
regexp-variables, all, none
NON LINT-CHECK OPTIONS
-u Package
BUGS
AUTHOR

B::O, O – Generic interface to Perl Compiler backends

SYNOPSIS
DESCRIPTION
CONVENTIONS
IMPLEMENTATION
BUGS
AUTHOR

B::Showlex – Show lexical variables used in functions or files

SYNOPSIS
DESCRIPTION
AUTHOR

B::Stackobj – Helper module for CC backend

SYNOPSIS
DESCRIPTION
AUTHOR

B::Stash – show what stashes are loaded

B::Terse – Walk Perl syntax tree, printing terse info about ops

SYNOPSIS
DESCRIPTION
AUTHOR

B::Xref – Generates cross reference reports for Perl programs

SYNOPSIS
DESCRIPTION
OPTIONS
-oFILENAME, -r, -d, -D[tO]

BUGS
AUTHOR

Bblock, B::Bblock – Walk basic blocks

SYNOPSIS
DESCRIPTION
 Functions
 find_leaders
AUTHOR

Benchmark – benchmark running times of Perl code

SYNOPSIS
DESCRIPTION
 Methods
 new, debug, iters
 Standard Exports
 timeit(COUNT, CODE), timethis (COUNT, CODE, [TITLE, [STYLE]]), timethese (COUNT, CODEHASHREF, [STYLE]), timediff (T1, T2), timestr (TIMEDIFF, [STYLE, [FORMAT]])
 Optional Exports
 clearcache (COUNT), clearallcache (), cmpthese (COUT, CODEHASHREF, [STYLE]),
 cmpthese (RESULTSHASHREF, [STYLE]), countit(TIME, CODE), disablecache (),
 enablecache (), timesum (T1, T2)
NOTES
EXAMPLES
INHERITANCE
CAVEATS
SEE ALSO
AUTHORS
MODIFICATION HISTORY

ByteLoader – load byte compiled perl code

SYNOPSIS
DESCRIPTION
AUTHOR
SEE ALSO

Bytecode, B::Bytecode – Perl compiler's bytecode backend

SYNOPSIS
DESCRIPTION
OPTIONS
 -ofilename, -afilename, —, -f, -fcompress-nullops, -fomit-sequence-numbers,
 -fbypass-nullops, -On, -D, -Do, -Db, -Da, -DC, -S, -upackage
EXAMPLES
BUGS
AUTHORS

CGI – Simple Common Gateway Interface Class

SYNOPSIS

ABSTRACT

DESCRIPTION

PROGRAMMING STYLE

CALLING CGI.PM ROUTINES

CREATING A NEW QUERY OBJECT (OBJECT-ORIENTED STYLE):

CREATING A NEW QUERY OBJECT FROM AN INPUT FILE

FETCHING A LIST OF KEYWORDS FROM THE QUERY:

FETCHING THE NAMES OF ALL THE PARAMETERS PASSED TO YOUR SCRIPT:

FETCHING THE VALUE OR VALUES OF A SINGLE NAMED PARAMETER:

SETTING THE VALUE(S) OF A NAMED PARAMETER:

APPENDING ADDITIONAL VALUES TO A NAMED PARAMETER:

IMPORTING ALL PARAMETERS INTO A NAMESPACE:

DELETING A PARAMETER COMPLETELY:

DELETING ALL PARAMETERS:

DIRECT ACCESS TO THE PARAMETER LIST:

FETCHING THE PARAMETER LIST AS A HASH:

SAVING THE STATE OF THE SCRIPT TO A FILE:

RETRIEVING CGI ERRORS

USING THE FUNCTION-ORIENTED INTERFACE

:cgi, :form, :html2, :html3, :html4, :netscape, :html, :standard, :all

PRAGMAS

-any, -compile, -nosticky, -no_undef_params, -no_xhtml, -nph, -newstyle_urls,
-oldstyle_urls, -autoload, -no_debug, -debug, -private_tempfiles

SPECIAL FORMS FOR IMPORTING HTML-TAG FUNCTIONS

1. start_table() (generates a <table tag), 2. end_table() (generates a </table tag), 3.
start_ul() (generates a <ul tag), 4. end_ul() (generates a </ul tag)

GENERATING DYNAMIC DOCUMENTS

CREATING A STANDARD HTTP HEADER:

GENERATING A REDIRECTION HEADER

CREATING THE HTML DOCUMENT HEADER

Parameters:, 4, 5, 6..

ENDING THE HTML DOCUMENT:

CREATING A SELF-REFERENCING URL THAT PRESERVES STATE INFORMATION:

OBTAINING THE SCRIPT'S URL

-absolute, -relative, -full, -path (-path_info), -query (-query_string), -base

MIXING POST AND URL PARAMETERS

CREATING STANDARD HTML ELEMENTS:

PROVIDING ARGUMENTS TO HTML SHORTCUTS

THE DISTRIBUTIVE PROPERTY OF HTML SHORTCUTS

HTML SHORTCUTS AND LIST INTERPOLATION

NON-STANDARD HTML SHORTCUTS

AUTOESCAPING HTML

```
$escaped_string = escapeHTML("unescaped string"); $charset =  
charset([$charset]); $flag = autoEscape([$flag]);
```

PRETTY-PRINTING HTML

CREATING FILL-OUT FORMS:

CREATING AN ISINDEX TAG

STARTING AND ENDING A FORM

application/x-www-form-urlencoded, multipart/form-data

CREATING A TEXT FIELD

Parameters

CREATING A BIG TEXT FIELD

CREATING A PASSWORD FIELD

CREATING A FILE UPLOAD FIELD

Parameters

CREATING A POPUP MENU

CREATING A SCROLLING LIST

Parameters:

CREATING A GROUP OF RELATED CHECKBOXES

Parameters:

CREATING A STANDALONE CHECKBOX

Parameters:

CREATING A RADIO BUTTON GROUP

Parameters:

CREATING A SUBMIT BUTTON

Parameters:

CREATING A RESET BUTTON

CREATING A DEFAULT BUTTON

CREATING A HIDDEN FIELD

Parameters:

CREATING A CLICKABLE IMAGE BUTTON

Parameters:

CREATING A JAVASCRIPT ACTION BUTTON

HTTP COOKIES

1. an expiration time, 2. a domain, 3. a path, 4. a "secure" flag, **-name**, **-value**, **-path**, **-domain**, **-expires**, **-secure**

WORKING WITH FRAMES

1. Create a <Frameset document, 2. Specify the destination for the document in the HTTP header, 3. Specify the destination for the document in the <form tag

LIMITED SUPPORT FOR CASCADING STYLE SHEETS

DEBUGGING

DUMPING OUT ALL THE NAME/VALUE PAIRS

FETCHING ENVIRONMENT VARIABLES

```
Accept(), raw_cookie(), user_agent(), path_info(), path_translated(),
remote_host(), script_name(), referer(), auth_type(), server_name(), virtual_host
(), server_port(), server_software(), remote_user(), user_name(), request_method(),
content_type(), http(), https()
```

USING NPH SCRIPTS

In the **use** statement, By calling the **nph()** method:, By using **-nph** parameters

Server Push

```
multipart_init(), multipart_start(), multipart_end(), multipart_final()
```

Avoiding Denial of Service Attacks

`$CGI::POST_MAX`, `$CGI::DISABLE_UPLOADS`, **1. On a script-by-script basis, 2. Globally for all scripts**

COMPATIBILITY WITH CGI-LIB.PL

AUTHOR INFORMATION

CREDITS

Matt Heffron (heffron@falstaff.css.beckman.com), James Taylor (james.taylor@srs.gov), Scott Anguish <sanguish@digifix.com>, Mike Jewell (mlj3u@virginia.edu), Timothy Shimmin (tes@kbs.citri.edu.au), Joergen Haegg (jh@axis.se), Laurent Delfosse (delfosse@delfosse.com), Richard Resnick (applepi1@aol.com), Craig Bishop (csb@barwonwater.vic.gov.au), Tony Curtis (tc@vcpc.univie.ac.at), Tim Bunce (Tim.Bunce@ig.co.uk), Tom Christiansen (tchrist@convex.com), Andreas Koenig (k@franz.wz.TU-Berlin.DE), Tim MacKenzie (Tim.MacKenzie@fulcrum.com.au), Kevin B. Hendricks (kbhend@dogwood.tyler.wm.edu), Stephen Dahmen (joyfire@inxpress.net), Ed Jordan (ed@fidalgo.net), David Alan Pisoni (david@cnaion.com), Doug MacEachern (dougmac@opengroup.org), Robin Houston (robin@oneworld.org), ...and many many more..

A COMPLETE EXAMPLE OF A SIMPLE FORM-BASED SCRIPT

BUGS

SEE ALSO

CGI::Apache – Backward compatibility module for CGI.pm

SYNOPSIS

ABSTRACT

DESCRIPTION

AUTHOR INFORMATION

BUGS

SEE ALSO

CGI::Carp, CGI::Carp – CGI routines for writing to the HTTPD (or

other) error log

SYNOPSIS

DESCRIPTION

REDIRECTING ERROR MESSAGES

MAKING PERL ERRORS APPEAR IN THE BROWSER WINDOW

Changing the default message

MAKING WARNINGS APPEAR AS HTML COMMENTS

CHANGE LOG

AUTHORS

SEE ALSO

CGI::Cookie – Interface to Netscape Cookies

SYNOPSIS

DESCRIPTION

USING CGI::Cookie

1. expiration date, 2. domain, 3. path, 4. secure flag

Creating New Cookies

Sending the Cookie to the Browser

Recovering Previous Cookies

Manipulating Cookies

`name()`, `value()`, `domain()`, `path()`, `expires()`

AUTHOR INFORMATION
BUGS
SEE ALSO

CGI::Fast – CGI Interface for Fast CGI

SYNOPSIS
DESCRIPTION
OTHER PIECES OF THE PUZZLE
WRITING FASTCGI PERL SCRIPTS
INSTALLING FASTCGI SCRIPTS
USING FASTCGI SCRIPTS AS CGI SCRIPTS
EXTERNAL FASTCGI SERVER INVOCATION
 FCGI_SOCKET_PATH, FCGI_LISTEN_QUEUE

CAVEATS
AUTHOR INFORMATION
BUGS
SEE ALSO

CGI::Pretty – module to produce nicely formatted HTML code

SYNOPSIS
DESCRIPTION
 Tags that won't be formatted
 Customizing the Indenting
BUGS
AUTHOR
SEE ALSO

CGI::Push – Simple Interface to Server Push

SYNOPSIS
DESCRIPTION
USING CGI::Push
 –next_page, –last_page, –type, –delay, –cookie, –target, –expires, –nph
 Heterogeneous Pages
 Changing the Page Delay on the Fly
INSTALLING CGI::Push SCRIPTS
AUTHOR INFORMATION
BUGS
SEE ALSO

CGI::Switch – Backward compatibility module for defunct CGI::Switch

SYNOPSIS
ABSTRACT
DESCRIPTION
AUTHOR INFORMATION
BUGS
SEE ALSO

CGI::Util – Internal utilities used by CGI module

SYNOPSIS
DESCRIPTION
AUTHOR INFORMATION

SEE ALSO

CPAN – query, download and build perl modules from CPAN sites

SYNOPSIS

DESCRIPTION

Interactive Mode

Searching for authors, bundles, distribution files and modules, make, test, install, clean modules or distributions, get, readme, look module or distribution, ls author, Signals

CPAN::Shell

autobundle

recompile

The four CPAN::* Classes: Author, Bundle, Module, Distribution

Programmer's interface

expand(\$type,@things), expandany(@things), Programming Examples

Methods in the other Classes

```
CPAN::Author::as_glimpse(), CPAN::Author::as_string(),
CPAN::Author::email(), CPAN::Author::fullname(),
CPAN::Author::name(), CPAN::Bundle::as_glimpse(),
CPAN::Bundle::as_string(), CPAN::Bundle::clean(),
CPAN::Bundle::contains(), CPAN::Bundle::force($method,@args),
CPAN::Bundle::get(), CPAN::Bundle::inst_file(),
CPAN::Bundle::inst_version(), CPAN::Bundle::uptodate(),
CPAN::Bundle::install(), CPAN::Bundle::make(),
CPAN::Bundle::readme(), CPAN::Bundle::test(),
CPAN::Distribution::as_glimpse(), CPAN::Distribution::as_string(),
CPAN::Distribution::clean(), CPAN::Distribution::containsmods(),
CPAN::Distribution::cvs_import(), CPAN::Distribution::dir(),
CPAN::Distribution::force($method,@args), CPAN::Distribution::get(),
CPAN::Distribution::install(), CPAN::Distribution::isa_perl(),
CPAN::Distribution::look(), CPAN::Distribution::make(),
CPAN::Distribution::prereq_pm(), CPAN::Distribution::readme(),
CPAN::Distribution::test(), CPAN::Distribution::uptodate(),
CPAN::Index::force_reload(), CPAN::Index::reload(),
CPAN::InfoObj::dump(), CPAN::Module::as_glimpse(),
CPAN::Module::as_string(), CPAN::Module::clean(),
CPAN::Module::cpan_file(), CPAN::Module::cpan_version(),
CPAN::Module::cvs_import(), CPAN::Module::description(),
CPAN::Module::force($method,@args), CPAN::Module::get(),
CPAN::Module::inst_file(), CPAN::Module::inst_version(),
CPAN::Module::install(), CPAN::Module::look(),
CPAN::Module::make(), CPAN::Module::manpage_headline(),
CPAN::Module::readme(), CPAN::Module::test(),
CPAN::Module::uptodate(), CPAN::Module::userid()
```

Cache Manager

Bundles

Prerequisites

Finding packages and VERSION

Debugging

Floppy, Zip, Offline Mode

CONFIGURATION

```
o conf <scalar option>, o conf <scalar option> <value>, o conf <list
option>, o conf <list option> [shift|pop], o conf <list option>
```

[unshift|push|splice] <list>

Note on urllist parameter's format

urllist parameter has CD-ROM support

SECURITY

EXPORT

POPULATE AN INSTALLATION WITH LOTS OF MODULES

WORKING WITH CPAN.pm BEHIND FIREWALLS

Three basic types of firewalls

http firewall, ftp firewall, One way visibility, SOCKS, IP Masquerade

Configuring lynx or ncftp for going through a firewall

FAQ

1), 2), 3), 4), 5), 6), 7), 8), 9), 10)

BUGS

AUTHOR

TRANSLATIONS

SEE ALSO

CPAN::FirstTime – Utility for CPAN::Config file Initialization

SYNOPSIS

DESCRIPTION

CPANox, CPAN::Nox – Wrapper around CPAN.pm without using any XS

module

SYNOPSIS

DESCRIPTION

SEE ALSO

Carp, carp – warn of errors (from perspective of caller)

SYNOPSIS

DESCRIPTION

Forcing a Stack Trace

BUGS

Carp::Heavy, Carp heavy machinery – no user serviceable parts inside

Class::ISA — report the search path for a class's ISA tree

SYNOPSIS

DESCRIPTION

FUNCTIONS

the function Class::ISA::super_path(\$CLASS) , the function

Class::ISA::self_and_super_path(\$CLASS) , the function

Class::ISA::self_and_super_versions(\$CLASS)

CAUTIONARY NOTES

COPYRIGHT

AUTHOR

Class::Struct – declare struct-like datatypes as Perl classes

SYNOPSIS

DESCRIPTION

The struct() function

Class Creation at Compile Time

Element Types and Accessor Methods

Scalar (`'$'` or `'*$'`), Array (`'@'` or `'*@'`), Hash (`'%'` or `'*%'`), Class (`'Class_Name'` or `'*Class_Name'`)

Initializing with new

EXAMPLES

Example 1, Example 2, Example 3

Author and Modification History

Config – access Perl configuration information

SYNOPSIS

DESCRIPTION

```
myconfig(), config_sh(), config_vars(@names)
```

EXAMPLE

WARNING

GLOSSARY

– _a, _exe, _o

- a afs, afsroot, alignbytes, ansi2knr, aphostname, api_revision, api_subversion, api_version, api_versionstring, ar, archlib, archlibexp, archname64, archname, archobjs, asctime_r_proto, awk
- b baserev, bash, bin, binexp, bison, byacc, byteorder
- c c, castflags, cat, cc, cccdlflags, ccdlflags, ccflags, ccflags_uselargefiles, ccname, ccsymbols, ccversion, cf_by, cf_email, cf_time, charsize, chgrp, chmod, chown, clocktype, comm, compress, contains, cp, cpio, cpp, cpp_stuff, cppccsymbols, cppflags, cpplast, cppminus, cpprun, cppstdin, cppsymbols, crypt_r_proto, cryptlib, csh, ctermid_r_proto, ctime_r_proto
- d d_fwalk, d_access, d_accessx, d_alarm, d_archlib, d_asctime_r, d_atofl, d_atoll, d_attribut, d_bcmp, d_bcopy, d_bsd, d_bsdgetpgrp, d_bsdsetpgrp, d_bzero, d_casti32, d_castneg, d_charvspr, d_chown, d_chroot, d_chsize, d_class, d_closedir, d_msgsghdr_s, d_const, d_crypt, d_crypt_r, d_csh, d_ctermid_r, d_ctime_r, d_cuserid, d_dbl_dig, d_dbminiproto, d_difftime, d_dirfd, d_dirnamlen, d_dlopen, d_dlsymun, d_dosuid, d_drand48_r, d_drand48proto, d_dup2, d_eaccess, d_endgrent, d_endgrent_r, d_endhrent, d_endhostent_r, d_endnrent, d_endnetent_r, d_endpent, d_endprotoent_r, d_endpwent, d_endpwent_r, d_endsent, d_endservent_r, d_eofnblk, d_eunice, d_fchdir, d_fchmod, d_fchown, d_fcntl, d_fcntl_can_lock, d_fd_macros, d_fd_set, d_fds_bits, d_fgetpos, d_finite, d_finitel, d_flexfnam, d_flock, d_flockproto, d_fork, d_fp_class, d_fpathconf, d_fpclass, d_fpclassify, d_fpclassl, d_fpos64_t, d_frexp, d_fs_data_s, d_fseeko, d_fsetpos, d_fstatfs, d_fstatvfs, d_fsync, d_ftello, d_ftime, d_Gconvert, d_getcwd, d_getespwnam, d_getfsstat, d_getgrent, d_getgrent_r, d_getgrgid_r, d_getgrnam_r, d_getgrps, d_gethbyaddr, d_gethbyname, d_gethent, d_gethname, d_gethostbyaddr_r, d_gethostbyname_r, d_gethostent_r, d_gethostprotos, d_getitimer, d_getlogin, d_getlogin_r, d_getmnt, d_getmntent, d_getnbyaddr, d_getnbyname, d_getnrent, d_getnetbyaddr_r, d_getnetbyname_r, d_getnetent_r, d_getnetprotos, d_getpagsz, d_getpbyname, d_getpbynumber, d_getpent, d_getpgid, d_getpgrp2, d_getpgrp, d_getppid, d_getprior, d_getprotobynam_r, d_getprotobynumber_r, d_getprotoent_r, d_getprototprotos,

d_getppwnam, d_getpwnam, d_getpwent, d_getpwent_r, d_getpwnam_r, d_getpwnuid_r, d_getsbyname, d_getsbyport, d_getsent, d_getservbyname_r, d_getservbyport_r, d_getservent_r, d_getservprotos, d_getspnam, d_getspnam_r, d_gettimeod, d_gmtime_r, d_gnulibc, d_grpasswd, d_hasmntopt, d_htonl, d_index, d_inetaton, d_int64_t, d_isascii, d_isfinite, d_isinf, d_isnan, d_isnanl, d_killpg, d_lchown, d_ldbl_dig, d_link, d_localtime_r, d_locconv, d_lockf, d_longdbl, d_longlong, d_lseekproto, d_lstat, d_madvise, d_mblen, d_mbstowcs, d_mbtowc, d_memchr, d_memcmp, d_memcpy, d_memmove, d_memset, d_mkdir, d_mkdtemp, d_mkfifo, d_mkstemp, d_mkstemps, d_mktime, d_mmap, d_modfl, d_modfl_pow32_bug, d_mprotect, d_msg, d_msg_ctrunc, d_msg_dontroute, d_msg_oob, d_msg_peek, d_msg_proxy, d_msgctl, d_msgget, d_msghdr_s, d_msgrcv, d_msgsnd, d_msync, d_munmap, d_mymalloc, d_nice, d_nl_langinfo, d_nv_preserves_uv, d_off64_t, d_old_pthread_create_joinable, d_oldpthreads, d_oldsock, d_open3, d_pathconf, d_pause, d_perl_otherlibdirs, d_phostname, d_pipe, d_poll, d_portable, d_PRIId64, d_PRIIdbl, d_PRIEldbl, d_PRIFldbl, d_PRIFUldbl, d_PRIGldbl, d_PRIGUldbl, d_PRIi64, d_PRIo64, d_PRIu64, d_PRIx64, d_PRIXU64, d_proclselfexe, d_pthread_atfork, d_pthread_yield, d_pwage, d_pwchange, d_pwclass, d_pwcomment, d_pwexpire, d_pwgecos, d_pwpasswd, d_pwquota, d_qgcvt, d_quad, d_random_r, d_readdir64_r, d_readdir, d_readdir_r, d_readlink, d_readv, d_recvmsg, d_rename, d_rewinddir, d_rmdir, d_safebcpy, d_safemcpy, d_sanemcmp, d_sbrkproto, d_sched_yield, d_scm_rights, d_SCNfldbl, d_seekdir, d_select, d_sem, d_semctl, d_semctl_semids, d_semctl_semuns, d_semget, d_semop, d_sendmsg, d_setegid, d_seteuid, d_setgrent, d_setgrent_r, d_setgrps, d_sethent, d_sethostent_r, d_setitimer, d_setlinebuf, d_setlocale, d_setlocale_r, d_setnent, d_setnetent_r, d_setpent, d_setpgid, d_setpgrp2, d_setpgrp, d_setprior, d_setproctitle, d_setprotoent_r, d_setpwent, d_setpwent_r, d_setregid, d_setresgid, d_setresuid, d_setreuid, d_setrgid, d_setruid, d_setsent, d_setservent_r, d_setsid, d_setvbuf, d_sfio, d_shm, d_shmat, d_shmatprototype, d_shmctl, d_shmdt, dshmget, d_sigaction, d_sigprocmask, d_sigsetjmp, d_socketatmark, d_socketatmarkproto, d_socket, d_socklen_t, d_socketpair, d_socks5_init, d_sqrtl, d_srand48_r, d_srandom_r, d_sresgproto, d_sresuproto, d_statblks, d_statfs_f_flags, d_statfs_s, d_statvfs, d_stdio_cnt_lval, d_stdio_ptr_lval, d_stdio_ptr_lval_nochange_cnt, d_stdio_ptr_lval_sets_cnt, d_stdio_stream_array, d_stdiobase, d_stdstdio, d_strchr, d_strcoll, d_strctcpy, d_strerror, d_strerror_r, d_strftime, d_strtod, d_strtol, d_strtold, d_strtoll, d_strtoq, d_strtoul, d_strtoull, d_strtouq, d_strxfrm, d_suidsafe, d_symlink, d_syscall, d_syscallproto, d_sysconf, d_sysernlst, d_syserrlst, d_system, d_tcgetpgrp, d_tcsetpgrp, d_telldir, d_telldirproto, d_time, d_times, d_tm_tm_gmtoff, d_tm_tm_zone, d_tmpnam_r, d_truncate, d_ttyname_r, d_tzname, d_u32align, d_ualarm, d_umask, d_uname, d_union_semuns, d_unordered, d_usleep, d_usleepproto, d_ustat, d_vendorarch, d_vendorbin, d_vendorlib, d_vfork, d_void_closedir, d_voidsig, d_voidtty, d_volatile, d_vprintf, d_wait4, d_waitpid, d_wcstombs, d_wctomb, d_writev, d_xenix, date, db_hashtype, db_prefixtype, db_version_major, db_version_minor, db_version_patch, defvoidused, dirententrytype, dlex, dlsrc, doublesize, drand01, drand48 r proto, dynamic ext

- e eagain, ebcddic, echo, egrep, emacs, endgrent_r_proto,
endhostent_r_proto, endnetent_r_proto, endprotoent_r_proto,
endpwent_r_proto, endservent_r_proto, eunicefix, exe_ext, expr,
extensions, extras
- f fflushall, fflushNULL, find, firstmakefile, flex, fpossize, fpostype,
freetype, from, full_ar, full_csh, full_sed
- g gccosandvers, gccversion, getgrent_r_proto, getgrgid_r_proto,
getgrnam_r_proto, gethostbyaddr_r_proto, gethostbyname_r_proto,
gethostent_r_proto, getlogin_r_proto, getnetbyaddr_r_proto,
getnetbyname_r_proto, getnetent_r_proto, getprotobyname_r_proto,
getprotobynumber_r_proto, getprotoent_r_proto, getpwent_r_proto,
getpwnam_r_proto, getpwuid_r_proto, getservbyname_r_proto,
getservbyport_r_proto, getservent_r_proto, getspnam_r_proto,
gidformat, gidsign, gidsize, gidtype, glibpth, gmake, gmtime_r_proto,
gnulibc_version, grep, groupcat, groupstype, gzip
- h h_fcntl, h_sysfile, hint, hostcat
- i i16size, i16type, i32size, i32type, i64size, i64type, i8size, i8type,
i_arpanet, i_bsdiectl, i_crypt, i_db, i_dbm, i_dirent, i_dld, i_dlfcn,
i_fcntl, i_float, i_fp, i_fp_class, i_gdbm, i_grp, i_ieee_fp, i_inttypes,
i_langinfo, i_libutil, i_limits, i_locale, i_machcthr, i_malloc, i_math,
i_memory, i_mntent, i_ndbm, i_netdb, i_neterro, i_netinettcp, i_niin,
i_poll, i_prot, i_pthread, i_pwd, i_rpcsvcd, i_sfio, i_sgTTY, i_shadow,
i_socks, i_stdarg, i_stddef, i_stdlib, i_string, i_sunmath, i_sysaccess,
i_sysdir, i_sysfile, i_sysfilio, i_sysin, i_sysioctl, i_syslog,
i_sysman, i_sysmode, i_sysmount, i_sysndir, i_sysparam, i_sysresrc,
i_syssecrt, i_sysselect, i_syssockio, i_sysstat, i_sysstatfs,
i_sysstatvfs, i_systime, i_systimek, i_systimes, i_systypes, i_sysuio,
i_sysun, i_sysutname, i_sysvfs, i_syswait, i_termio, i_termios, i_time,
i_unistd, i_ustat, i_utime, i_values, i_varargs, i_varhdr, i_vfork,
ignore_versioned_solibs, inc_version_list, inc_version_list_init,
incpath, inews, installarchlib, installbin, installmanldir,
installman3dir, installprefix, installprefixexp, installprivlib,
installscript, installssearch, installssearchbin, installssearchlib,
installstyle, installusrbinperl, installvendorarch, installvendorbin,
installvendorlib, intsize, issymlink, ivdformat, ivsize, ivtype
- k known_extensions, ksh
- l ld, lddlflags, ldflags, ldflags_uselargefiles, ldlibpthname, less,
lib_ext, libc, libperl, libpth, libs, libsdirs, libsfiles, libsfound,
libspath, libswanted, libswanted_uselargefiles, line, lint, lkflags, ln,
lns, localtime_r_proto, locincpth, loclibpth, longdblsize,
longlongsize, longsize, lp, lpr, ls, lseeksize, lseektype
- m mail, mailx, make, make_set_make, mallocobj, mallocsrc, malloctype,
manldir, manldirexp, man1ext, man3dir, man3direxp, man3ext
- M Mcc, mips_type, mkdir, mmaptype, modetype, more, multiarch, mv,
myarchname, mydomain, myhostname, myuname
- n n, need_va_copy, netdb_hlen_type, netdb_host_type, netdb_name_type,
netdb_net_type, nm, nm_opt, nm_so_opt, nonxs_ext, nroff,
nv_preserves_uv_bits, nveformat, nvEUformat, nvffformat, nvFUformat,
nvGformat, nvGUformat, nvsize, nvtype

- o** o_nonblock, obj_ext, old_pthread_create_joinable, optimize, orderlib, osname, osvers, otherlibdirs
- p** package, pager, passcat, patchlevel, path_sep, perl5, perl, perl_patchlevel
- P** PERL_REVISION, PERL_SUBVERSION, PERL_VERSION, perladmin, perllibs, perlpath, pg, phostname, pidtype, plibpth, pm_apiversion, pmake, pr, prefix, prefixexp, privlib, privlibexp, procseluxe, prototype, ptrsize
- q** quadkind, quadtype
- r** randbits, randfunc, random_r_proto, randseedtype, ranlib, rd_nodata, readdir64_r_proto, readdir_r_proto, revision, rm, rmail, run, runnm
- s** sched_yield, scriptdir, scriptdirexp, sed, seedfunc, selectminbits, selecttype, sendmail, setgrent_r_proto, sethostent_r_proto, setlocale_r_proto, setnetent_r_proto, setprotoent_r_proto, setpwent_r_proto, setserverent_r_proto, sh, shar, sharpbang, shmatttype, shortsize, shrpenv, shsharp, sig_count, sig_name, sig_name_init, sig_num, sig_num_init, sig_size, signal_t, sitearch, sitearchexp, sitebin, sitebinexp, sitelib, sitelib_stem, sitelibexp, siteprefix, siteprefixexp, sizesize, sizetype, sleep, smail, so, sockethdr, socketlib, socksize, sort, spackage, spitshell, sPRId64, sPRIeldbl, sPRIEUldbl, sPRIfldbl, sPRIFUldbl, sPRIGldbl, sPRIGUldbl, sPRIi64, sPRIo64, sPRIu64, sPRIx64, sPRIXU64, srand48_r_proto, srandom_r_proto, src, sSCNfldbl, ssize, startperl, startsh, static_ext, stdchar, stdio_base, stdio_bufsiz, stdio_cnt, stdio_filbuf, stdio_ptr, stdio_stream_array, strerror_r_proto, strings, submit, subversion, sysman
- t** tail, tar, targetarch, tbl, tee, test, timeincl, timetype, tmpnam_r_proto, to, touch, tr, trnl, troff, ttyname_r_proto
- u** u16size, u16type, u32size, u32type, u64size, u64type, u8size, u8type, uidformat, uidsign, uidsize, uidtype, uname, uniq, uquadtype, use5005threads, use64bitall, use64bitint, usecrosscompile, usedl, useithreads, uselargefiles, uselongdouble, usemorebits, usemultiplicity, usemymalloc, usernm, useopcode, useperlio, useposix, usereentrant, usesfio, useshrplib, usesocks, usethreads, usevendorprefix, usevfork, usrinc, uuname, uvoformat, uvsize, uvtype, uvuformat, uvxformat, uvXUformat
- v** vendorarch, vendorarchexp, vendorbin, vendorbinexp, vendorlib, vendorlib_stem, vendorlibexp, vendorprefix, vendorprefixexp, version, version_patchlevel_string, versiononly, vi, voidflags
- x** xlibpth, xs_apiversion
- y** yacc, yaccflags
- z** zcat, zip

NOTE

Cwd – get pathname of current working directory

SYNOPSIS

DESCRIPTION

getcwd and friends
 getcwd, cwd, fastcwd, fastgetcwd

abs_path and friends
 abs_path, realpath, fast_abs_path

\$ENV{PWD}

NOTES

SEE ALSO

DB – programmatic interface to the Perl debugging API (draft, subject to change)

SYNOPSIS

DESCRIPTION

Global Variables

\$DB::sub, %DB::sub, \$DB::single, \$DB::signal, \$DB::trace, @DB::args,
 @DB::dbline, %DB::dbline, \$DB::package, \$DB::filename, \$DB::subname,
 \$DB::lineno

API Methods

CLIENT-register(), CLIENT-evalcode(String), CLIENT-skippkg('D::hide'),
 CLIENT-run(), CLIENT-step(), CLIENT-next(), CLIENT-done()

Client Callback Methods

CLIENT-init(), CLIENT-prestop([String]), CLIENT-stop(), CLIENT-idle(),
 CLIENT-poststop([String]), CLIENT-evalcode(String), CLIENT-cleanup(),
 CLIENT-output(List)

BUGS

AUTHOR

DB_File – Perl5 access to Berkeley DB version 1.x

SYNOPSIS

DESCRIPTION

DB_HASH, DB_BTREE, DB_RECNO

Using DB_File with Berkeley DB version 2 or greater
 Interface to Berkeley DB
 Opening a Berkeley DB Database File
 Default Parameters
 In Memory Databases

DB_HASH

A Simple Example

DB_BTREE

Changing the BTree sort order
 Handling Duplicate Keys
 The get_dup() Method
 The find_dup() Method
 The del_dup() Method
 Matching Partial Keys

DB_RECNO

The 'bval' Option
 A Simple Example
 Extra RECNO Methods

\$X->push(list) ;, \$value = \$X->pop ;, \$X->shift, \$X->unshift(list) ;,
 \$X->length, \$X->splICE(offset, length, elements);

Another Example

THE API INTERFACE

```
$status = $X->get($key, $value [, $flags]) ;, $status = $X->put($key, $value
[, $flags]) ;, $status = $X->del($key [, $flags]) ;, $status = $X->fd ;, $status =
$X->seq($key, $value, $flags) ;, $status = $X->sync([ $flags]) ;
```

DBM FILTERS

filter_store_key, filter_store_value, filter_fetch_key, filter_fetch_value

The Filter

An Example — the NULL termination problem.

Another Example — Key is a C int.

HINTS AND TIPS

Locking: The Trouble with fd

Safe ways to lock a database

Tie::DB_Lock, Tie::DB_LockFile, DB_File::Lock

Sharing Databases With C Applications

The `untie()` Gotcha

COMMON QUESTIONS

Why is there Perl source in my database?

How do I store complex data structures with DB_File?

What does "Invalid Argument" mean?

What does "Bareword 'DB_File' not allowed" mean?

REFERENCES

HISTORY

BUGS

AVAILABILITY

COPYRIGHT

SEE ALSO

AUTHOR

Data::Dumper – stringified perl data structures, suitable for both

printing and eval

SYNOPSIS

DESCRIPTION

Methods

PACKAGE-new(*ARRAYREF* [, *ARRAYREF*]), *\$OBJ*-Dump or
PACKAGE-Dump(*ARRAYREF* [, *ARRAYREF*]), *\$OBJ*-Seen(*[HASHREF]*),
\$OBJ-Values(*[ARRAYREF]*), *\$OBJ*-Names(*[ARRAYREF]*), *\$OBJ*-Reset

Functions

Dumper(*LIST*)

Configuration Variables or Methods

\$Data::Dumper::Indent or *\$OBJ*-Indent(*[NEWVAL]*), *\$Data::Dumper::Purity*
or *\$OBJ*-Purity(*[NEWVAL]*), *\$Data::Dumper::Pad* or *\$OBJ*-Pad(*[NEWVAL]*),
\$Data::Dumper::Varname or *\$OBJ*-Varname(*[NEWVAL]*),
\$Data::Dumper::Useqq or *\$OBJ*-Useqq(*[NEWVAL]*), *\$Data::Dumper::Terse* or
\$OBJ-Terse(*[NEWVAL]*), *\$Data::Dumper::Freezer* or
\$OBJ-Freezer(*[NEWVAL]*), *\$Data::Dumper::Toaster* or
\$OBJ-Toaster(*[NEWVAL]*), *\$Data::Dumper::Deepcopy* or
\$OBJ-Deepcopy(*[NEWVAL]*), *\$Data::Dumper::Quotekeys* or
\$OBJ-Quotekeys(*[NEWVAL]*), *\$Data::Dumper::Bless* or *\$OBJ*-Bless(*[NEWVAL]*),
\$Data::Dumper::Maxdepth or *\$OBJ*-Maxdepth(*[NEWVAL]*),


```
$Data::Dumper::Useperl or $OBJ-Useperl([NEWVAL]),  
$Data::Dumper::Sortkeys or $OBJ-Sortkeys([NEWVAL]),  
$Data::Dumper::Deparse or $OBJ-Deparse([NEWVAL])
```

Exports

Dumper

EXAMPLES

BUGS

AUTHOR

VERSION

SEE ALSO

Devel::DProf – a Perl code profiler

SYNOPSIS

DESCRIPTION

PROFILE FORMAT

AUTOLOAD

ENVIRONMENT

BUGS

SEE ALSO

Devel::PPPort, Perl/Pollution/Portability

SYNOPSIS

DESCRIPTION

WriteFile

ppport.h

AUTHOR

SEE ALSO

Devel::Peek – A data debugging tool for the XS programmer

SYNOPSIS

DESCRIPTION

Runtime debugging

Memory footprint debugging

EXAMPLES

A simple scalar string

A simple scalar number

A simple scalar with an extra reference

A reference to a simple scalar

A reference to an array

A reference to a hash

Dumping a large array or hash

A reference to an SV which holds a C pointer

A reference to a subroutine

EXPORTS

BUGS

AUTHOR

SEE ALSO

Devel::SelfStubber – generate stubs for a SelfLoading module

SYNOPSIS

DESCRIPTION

Digest:: – Modules that calculate message digests

SYNOPSIS

DESCRIPTION

binary, hex, base64

OO INTERFACE

```
$ctx = Digest-XXX($arg, ...), $ctx = Digest-new(XXX = $arg, ...), $ctx =
Digest::XXX-new($arg, ...), $ctx-reset, $ctx-add($data,...), $ctx-addfile($io_handle),
$ctx-digest, $ctx-hexdigest, $ctx-b64digest
```

SEE ALSO

AUTHOR

Digest::MD5 – Perl interface to the MD5 Algorithm

SYNOPSIS

DESCRIPTION

FUNCTIONS

```
md5($data, ...), md5_hex($data, ...), md5_base64($data, ...)
```

METHODS

```
$md5 = Digest::MD5-new, $md5-reset, $md5-add($data,...), $md5-addfile($io_handle),
$md5-digest, $md5-hexdigest, $md5-b64digest
```

EXAMPLES

SEE ALSO

COPYRIGHT

AUTHORS

DirHandle – supply object methods for directory handles

SYNOPSIS

DESCRIPTION

NOTES

Dumpvalue – provides screen dump of Perl data.

SYNOPSIS

DESCRIPTION

Creation

```
arrayDepth, hashDepth, compactDump, veryCompact, globPrint,
dumpDBFiles, dumpPackages, dumpReused, tick, quoteHighBit, printUndef,
usageOnly, unctrl, subdump, bareStringify, quoteHighBit, stopDbSignal
```

Methods

```
dumpValue, dumpValues, stringify, dumpvars, set_quote, set_unctrl, compactDump,
veryCompact, set, get
```

DynaLoader – Dynamically load C libraries into Perl code

SYNOPSIS

DESCRIPTION

```
@dl_library_path, @dl_resolve_using, @dl_require_symbols, @dl_librefs, @dl_modules,
dl_error(), $dl_debug, dl_findfile(), dl_expandspec(), dl_load_file(),
dl_unload_file(), dl_loadflags(), dl_find_symbol(),
dl_find_symbol_anywhere(), dl_undef_symbols(), dl_install_xsub(),
bootstrap()
```

AUTHOR

DynaLoader::XSLoader, XSLoader – Dynamically load C libraries into

Perl code

SYNOPSIS

DESCRIPTION

AUTHOR

Encode – character encodings

SYNOPSIS

Table of Contents

DESCRIPTION

TERMINOLOGY

PERL ENCODING API

```
$octets = encode(ENCODING, $string [, CHECK]), $string = decode(ENCODING,
$octets [, CHECK]), [$length =] from_to($octets, FROM_ENC, TO_ENC [, CHECK]),
$octets = encode_utf8($string) ; , $string = decode_utf8($octets [, CHECK]);
```

Listing available encodings

Defining Aliases

Encoding via PerlIO

Handling Malformed Data

```
CHECK = Encode::FB_DEFAULT ( == 0), CHECK = Encode::FB_CROAK ( == 1), CHECK =
Encode::FB_QUIET, CHECK = Encode::FB_WARN, perlqq mode (CHECK =
Encode::FB_PERLQQ), HTML charref mode (CHECK = Encode::FB_HTMLCREF), XML charref
mode (CHECK = Encode::FB_XMLCREF), The bitmask
```

Unimplemented fallback schemes

Defining Encodings

The UTF-8 flag

Goal #1:, Goal #2:, Goal #3:, Goal #4:

Messing with Perl's Internals

```
is_utf8(String [, CHECK]), _utf8_on(String), _utf8_off(String)
```

SEE ALSO

MAINTAINER

Encode::Alias – alias definitions to encodings

SYNOPSIS

DESCRIPTION

As a simple string, As a qr// compiled regular expression, e.g.:, As a code reference, e.g.:

Alias overloading

SEE ALSO

Encode::Byte – Single Byte Encodings

SYNOPSIS

ABSTRACT

DESCRIPTION

SEE ALSO

Encode::CJKConstants — Internally used by Encode::??::ISO_2022_*

Encode::CN – China-based Chinese Encodings

SYNOPSIS
DESCRIPTION
NOTES
BUGS
SEE ALSO

Encode::CN::HZ — internally used by Encode::CN

Encode::Config — internally used by Encode

Encode::EBCDIC – EBCDIC Encodings

SYNOPSIS
ABSTRACT
DESCRIPTION
SEE ALSO

Encode::Encoding – Encode Implementation Base Class

SYNOPSIS
DESCRIPTION

Methods you should implement

`->encode($string [, $check]) , ->decode($octets [, $check])`

Other methods defined in Encode::Encodings

`->name, ->new_sequence, ->perlio_ok(), ->needs_lines()`

Example: Encode::ROT13

Why the heck Encode API is different?

Compiled Encodings

SEE ALSO

Scheme 1, Scheme 2, Other Schemes

Encode::Guess — Guesses encoding from data

SYNOPSIS
ABSTRACT
DESCRIPTION

`Encode::Guess-set_suspects, Encode::Guess-add_suspects, Encode::decode("Guess" ...),
Encode::Guess-guess($data) , guess_encoding($data, [, list of suspects])`

CAVEATS

TO DO

SEE ALSO

Encode::JP – Japanese Encodings

SYNOPSIS
ABSTRACT
DESCRIPTION
Note on ISO-2022-JP(-1)?
BUGS
SEE ALSO

Encode::JP::H2Z — internally used by Encode::JP::2022_JP*

Encode::JP::JIS7 — internally used by Encode::JP

Encode::KR – Korean Encodings

SYNOPSIS

DESCRIPTION
BUGS
SEE ALSO

Encode::KR::2022_KR — internally used by Encode::KR

Encode::MIME::Header — MIME ‘B’ and ‘Q’ header encoding

SYNOPSIS
ABSTRACT
DESCRIPTION
BUGS
SEE ALSO

Encode::PerlIO — a detailed document on Encode and PerlIO

Overview
How does it work?
BUGS
 Workaround
 How can I tell whether my encoding fully supports PerlIO ?
SEE ALSO

Encode::Supported — Encodings supported by Encode

DESCRIPTION
 Encoding Names
Supported Encodings
 Built-in Encodings
 Encode::Unicode — other Unicode encodings
 Encode::Byte — Extended ASCII
 ISO-8859 and corresponding vendor mappings, KOI8 – De Facto Standard for the Cyrillic world, gsm0338 – Hentai Latin 1
 CJK: Chinese, Japanese, Korean (Multibyte)
 Encode::CN — Continental China, Encode::JP — Japan, Encode::KR — Korea, Encode::TW — Taiwan, Encode::HanExtra — More Chinese via CPAN, Encode::JIS2K — JIS X 0213 encodings via CPAN
 Miscellaneous encodings
 Encode::EBCDIC, Encode::Symbols, Encode::MIME::Header, Encode::Guess

Unsupported encodings

 ISO-2022-JP-2 [RFC1554], ISO-2022-CN [RFC1922], Various HP-UX encodings, Cyrillic encoding ISO-IR-111, ISO-8859-8-1 [Hebrew], ISIRI 3342, Iran System, ISIRI 2900 [Farsi], Thai encoding TCVN, Vietnamese encodings VPS, Various Mac encodings, (Mac) Indic encodings

Encoding vs. Charset — terminology

Encoding Classification (by Anton Tagunov and Dan Kogai)

 Microsoft-related naming mess

 KS_C_5601-1987, GB2312, Big5, Shift_JIS

Glossary

 character repertoire, coded character set (CCS), character encoding scheme (CES), charset (in MIME context), EUC, ISO-2022, UCS, UCS-2, Unicode, UTF, UTF-16

See Also

References

ECMA, ECMA-035 (eq ISO-2022), IANA, Assigned Charset Names by IANA, ISO, RFC, UC, Unicode Glossary

Other Notable Sites

czyborra.com, CJK.inf, Jungshik Shin's Hangul FAQ, debian.org: "Introduction to i18n"

Offline sources

CJKV Information Processing by Ken Lunde

Encode::Symbol – Symbol Encodings

SYNOPSIS

ABSTRACT

DESCRIPTION

SEE ALSO

Encode::TW – Taiwan-based Chinese Encodings

SYNOPSIS

DESCRIPTION

NOTES

BUGS

SEE ALSO

Encode::Unicode — Various Unicode Transformation Formats

SYNOPSIS

ABSTRACT

<http://www.unicode.org/glossary/> says:, Quick Reference

Size, Endianness, and BOM

by size

by endianness

BOM as integer when fetched in network byte order

Surrogate Pairs

SEE ALSO

Encode::lib::Encode::Alias, Encode::Alias – alias definitions to

encodings

SYNOPSIS

DESCRIPTION

As a simple string, As a qr// compiled regular expression, e.g.:. As a code reference, e.g.:

Alias overloading

SEE ALSO

Encode::lib::Encode::CJKConstants, Encode::CJKConstants.pm —

Internally used by Encode::??::ISO_2022_*

Encode::lib::Encode::CN::HZ, Encode::CN::HZ — internally used by

Encode::CN

Encode::lib::Encode::Config, Encode::Config — internally used by

Encode

Encode::lib::Encode::Encoding, Encode::Encoding – Encode

Implementation Base Class

SYNOPSIS

DESCRIPTION

Methods you should implement

```
->encode($string[, $check]) , ->decode($octets[, $check])
```

Other methods defined in Encode::Encodings

```
->name, ->new_sequence, ->perlio_ok(), ->needs_lines()
```

Example: Encode::ROT13

Why the heck Encode API is different?

Compiled Encodings

SEE ALSO

Scheme 1, Scheme 2, Other Schemes

Encode::lib::Encode::Guess, Encode::Guess — Guesses encoding from

data

SYNOPSIS

ABSTRACT

DESCRIPTION

```
Encode::Guess-set_suspects, Encode::Guess-add_suspects, Encode::decode("Guess" ...),
Encode::Guess-guess($data) , guess_encoding($data, [, list of suspects])
```

CAVEATS

TO DO

SEE ALSO

Encode::lib::Encode::JP::H2Z, Encode::JP::H2Z — internally used by

Encode::JP::2022_JP*

Encode::lib::Encode::JP::JIS7, Encode::JP::JIS7 — internally used

by Encode::JP

Encode::lib::Encode::KR::2022_KR, Encode::KR::2022_KR — internally

used by Encode::KR

Encode::lib::Encode::MIME::Header, Encode::MIME::Header — MIME ‘B’

and ‘Q’ header encoding

SYNOPSIS

ABSTRACT

DESCRIPTION

BUGS

SEE ALSO

Encode::lib::Encode::PerlIO, Encode::PerlIO — a detailed document

on Encode and PerlIO

Overview

How does it work?

BUGS

Workaround

How can I tell whether my encoding fully supports PerlIO ?

SEE ALSO

Encode::lib::Encode::Supported, Encode::Supported — Encodings

supported by Encode

DESCRIPTION

Encoding Names

Supported Encodings

Built-in Encodings

Encode::Unicode — other Unicode encodings

Encode::Byte — Extended ASCII

ISO-8859 and corresponding vendor mappings, KOI8 – De Facto Standard for the Cyrillic world, gsm0338 – Hentai Latin 1

CJK: Chinese, Japanese, Korean (Multibyte)

Encode::CN — Continental China, Encode::JP — Japan, Encode::KR — Korea, Encode::TW — Taiwan, Encode::HanExtra — More Chinese via CPAN, Encode::JIS2K — JIS X 0213 encodings via CPAN

Miscellaneous encodings

Encode::EBCDIC, Encode::Symbols, Encode::MIME::Header, Encode::Guess

Unsupported encodings

ISO-2022-JP-2 [RFC1554], ISO-2022-CN [RFC1922], Various HP-UX encodings, Cyrillic encoding ISO-IR-111, ISO-8859-8-1 [Hebrew], ISIRI 3342, Iran System, ISIRI 2900 [Farsi], Thai encoding TCVN, Vietnamese encodings VPS, Various Mac encodings, (Mac) Indic encodings

Encoding vs. Charset — terminology

Encoding Classification (by Anton Tagunov and Dan Kogai)

Microsoft-related naming mess

KS_C_5601-1987, GB2312, Big5, Shift_JIS

Glossary

character repertoire, coded character set (CCS), character encoding scheme (CES), charset (in MIME context), EUC, ISO-2022, UCS, UCS-2, Unicode, UTF, UTF-16

See Also

References

ECMA, ECMA-035 (eq ISO-2022), IANA, Assigned Charset Names by IANA, ISO, RFC, UC, Unicode Glossary

Other Notable Sites

czyborra.com, CJK.inf, Jungshik Shin's Hangul FAQ, debian.org: "Introduction to i18n"

Offline sources

CJKV Information Processing by Ken Lunde

Encode::lib::Encoder, Encode::Encoder — Object Oriented Encoder

SYNOPSIS

```
use Encode::Encoder;
# Encode::encode("ISO-8859-1", $data);
Encode::Encoder->new($data)->iso_8859_1; # OOP way
# shortcut
use Encode::Encoder qw(encoder);
encoder($data)->iso_8859_1;
# you can stack them!
```



```

encoder($data)->iso_8859_1->base64; # provided base64() is defined
# you can use it as a decoder as well
encoder($base64)->bytes('base64')->latin1;
# stringified
print encoder($data)->utf8->latin1; # prints the string in latin1
# numified
encoder("\x{abcd}\x{ef}g")->utf8 == 6; # true. bytes::length($data)

```

ABSTRACT**Description****Predefined Methods**

```

$e = Encode::Encoder->new([$data, $encoding]);, encoder(),
$e->data([$data]), $e->encoding([$encoding]),
$e->bytes([$encoding])

```

Example: base64 transcoder

Operator Overloading

SEE ALSO

Encodencoding, encoding – allows you to write your script in

non-ascii or non-utf8

SYNOPSIS**ABSTRACT****USAGE**

use encoding [*ENCNAME*] ;, use encoding *ENCNAME* [*STDIN => ENCNAME_IN ...*] ;, no encoding;

CAVEATS

NOT SCOPED

DO NOT MIX MULTIPLE ENCODINGS

Non-ASCII Identifiers and Filter option

use encoding *ENCNAME* Filter=>1;

EXAMPLE – Greekperl

KNOWN PROBLEMS

SEE ALSO

Encoder, Encode::Encoder — Object Oriented Encoder**SYNOPSIS**

```

use Encode::Encoder;
# Encode::encode("ISO-8859-1", $data);
Encode::Encoder->new($data)->iso_8859_1; # OOP way
# shortcut
use Encode::Encoder qw(encoder);
encoder($data)->iso_8859_1;
# you can stack them!
encoder($data)->iso_8859_1->base64; # provided base64() is defined
# you can use it as a decoder as well
encoder($base64)->bytes('base64')->latin1;
# stringified
print encoder($data)->utf8->latin1; # prints the string in latin1
# numified
encoder("\x{abcd}\x{ef}g")->utf8 == 6; # true. bytes::length($data)

```

ABSTRACT

Description

Predefined Methods

```
$e = Encode::Encoder->new([$data, $encoding]);, encoder(),
$e->data([$data]), $e->encoding([$encoding]),
$e->bytes([$encoding])
```

Example: base64 transcoder

Operator Overloading

SEE ALSO

English – use nice English (or awk) names for ugly punctuation

variables

SYNOPSIS

DESCRIPTION

PERFORMANCE

Env – perl module that imports environment variables as scalars or

arrays

SYNOPSIS

DESCRIPTION

LIMITATIONS

AUTHOR

Errno – System errno constants

SYNOPSIS

DESCRIPTION

CAVEATS

AUTHOR

COPYRIGHT

Exporter – Implements default import method for modules

SYNOPSIS

DESCRIPTION

How to Export

Selecting What To Export

How to Import

```
use ModuleName;,use ModuleName ();,use ModuleName qw(...);
```

Advanced features

Specialised Import Lists

Exporting without using Exporter's import method

Module Version Checking

Managing Unknown Symbols

Tag Handling Utility Functions

Generating combined tags

AUTOLOADED Constants

Exporter::Heavy – Exporter guts

SYNOPSIS

DESCRIPTION

ExtUtils::Command – utilities to replace common UNIX commands in

Makefiles etc.

SYNOPSIS

DESCRIPTION

cat

eqtime src dst

rm_rf files...

rm_f files...

touch files ..

mv source... destination

cp source... destination

chmod mode files..

mkpath directory..

test_f file

BUGS

SEE ALSO

AUTHOR

ExtUtils::Command::MM – Commands for the MM's to use in Makefiles

SYNOPSIS

DESCRIPTION

test_harness

ExtUtils::Constant – generate XS code to import C header constants

SYNOPSIS

DESCRIPTION

USAGE

IV, UV, NV, PV, PVN, SV, YES, NO, UNDEF

FUNCTIONS

C_stringify NAME

perl_stringify NAME

constant_types

memEQ_clause NAME, CHECKED_AT, INDENT

assign INDENT, TYPE, PRE, POST, VALUE..

return_clause

switch_clause INDENT, NAMELEN, ITEMHASH, ITEM..

params WHAT

dump_names

dogfood

C_constant, name, type, value, macro, default, pre, post, def_pre =item def_post, utf8

XS_constant PACKAGE, TYPES, SUBNAME, C_SUBNAME

autoload PACKAGE, VERSION, AUTOLOADER

WriteMakefileSnippet

WriteConstants ATTRIBUTE => VALUE [, ...], NAME, DEFAULT_TYPE, BREAKOUT_AT, NAMES, C_FILE, XS_FILE, SUBNAME, C_SUBNAME

AUTHOR

ExtUtils::Embed – Utilities for embedding Perl in C/C++ applications

SYNOPSIS

DESCRIPTION

@EXPORT

FUNCTIONS

xsinit(), Examples, ldopts(), Examples, perl_inc(), ccflags(), ccdlflags(),
ccopts(), xsi_header(), xsi_protos(@modules), xsi_body(@modules)

EXAMPLES

SEE ALSO

AUTHOR

ExtUtils::Install – install files from here to there

SYNOPSIS

DESCRIPTION

ExtUtils::Installed – Inventory management of installed modules

SYNOPSIS

DESCRIPTION

USAGE

FUNCTIONS

new(), modules(), files(), directories(), directory_tree(), validate(),
packlist(), version()

EXAMPLE

AUTHOR

ExtUtils::Liblist – determine libraries to use and how to use them

SYNOPSIS

DESCRIPTION

For static extensions, For dynamic extensions at build/link time, For dynamic extensions at load time

EXTRALIBS

LDLOADLIBS and LD_RUN_PATH

BSLOADLIBS

PORTABILITY

VMS implementation

Win32 implementation

SEE ALSO

ExtUtils::MM – OS adjusted ExtUtils::Maker subclass

SYNOPSIS

DESCRIPTION

ExtUtils::MM_Any – Platform agnostic MM methods

SYNOPSIS

DESCRIPTION

Inherently Cross-Platform Methods

File::Spec wrappers **DEPRECATED**

canonpath

catdir

catfile

curdir

file_name_is_absolute

path

rootdir

updir

Thought To Be Cross-Platform Methods

test_via_harness

test_via_script

AUTHOR

ExtUtils::MM_BeOS – methods to override UN*X behaviour in

ExtUtils::MakerMaker

SYNOPSIS

DESCRIPTION

perl_archive

ExtUtils::MM_Cygwin – methods to override UN*X behaviour in

ExtUtils::MakerMaker

SYNOPSIS

DESCRIPTION

canonpath, cflags, manifypods, perl_archive

ExtUtils::MM_DOS – DOS specific subclass of ExtUtils::MM_Unix

SYNOPSIS

DESCRIPTION

Overridden methods

replace_manpage_separator

AUTHOR

SEE ALSO

ExtUtils::MM_MacOS – methods to override UN*X behaviour in

ExtUtils::MakerMaker

SYNOPSIS

DESCRIPTION

maybe_command

guess_name

macify
patternify
init_main
init_others
init_dirscan
libscan (o)
constants (o)
static (o)
dlsyms (o)
dynamic (o)
clean (o)
realclean (o)
rulez (o)
processPL (o)

ExtUtils::MM_NW5 – methods to override UN*X behaviour in

ExtUtils::MakeMaker

SYNOPSIS
DESCRIPTION

constants (o)
static_lib (o)
dynamic_lib (o)

ExtUtils::MM_OS2 – methods to override UN*X behaviour in

ExtUtils::MakeMaker

SYNOPSIS
DESCRIPTION
METHODS

perl_archive_after

ExtUtils::MM_UWIN – U/WIN specific subclass of ExtUtils::MM_Unix

SYNOPSIS
DESCRIPTION
 Overridden methods
 replace_manpage_separator

AUTHOR
SEE ALSO

ExtUtils::MM_Unix – methods used by ExtUtils::MakeMaker

SYNOPSIS
DESCRIPTION
METHODS

SelfLoaded methods

- c_o (o)
- cflags (o)
- clean (o)
- const_cccmd (o)
- const_config (o)
- const_loadlibs (o)
- constants (o)
- depend (o)
- dir_target (o)
- dist (o)
- dist_basics (o)
- dist_ci (o)
- dist_core (o)
- dist_dir
- dist_test
- dlsyms (o)
- dynamic (o)
- dynamic_bs (o)
- dynamic_lib (o)
- exescan
- extliblist
- find_perl
- find_tests

Methods to actually produce chunks of text for the Makefile

- fixin
- force (o)
- guess_name
- has_link_code
- init_dirscan
- init_main
- init_others
- init_INST
- init_INSTALL
- init_lib2arch
- init_PERL
- init_PERM

install (o)
installbin (o)
libscan (o)
linkext (o)
lsdir
macro (o)
makeaperl (o)
makefile (o)
manifypods (o)
maybe_command
maybe_command_in_dirs
needs_linking (o)
nicetext
parse_abstract
parse_version
psthru (o)
perl_script
perldepend (o)
perm_rw (o)
perm_rwx (o)
pm_to_blib
post_constants (o)
post_initialize (o)
postamble (o)
ppd
prefixify
processPL (o)
quote_paren
realclean (o)
replace_manpage_separator
static (o)
static_lib (o)
staticmake (o)
subdir_x (o)
subdirs (o)
test (o)

test_via_harness (override)
test_via_script (override)
tool_autosplit (o)
tools_other (o)
tool_xsubpp (o)
top_targets (o)
writedoc
xs_c (o)
xs_cpp (o)
xs_o (o)
perl_archive
perl_archive_after
export_list

SEE ALSO

ExtUtils::MM_VMS – methods to override UN*X behaviour in

ExtUtils::MakeMaker

SYNOPSIS

DESCRIPTION

Methods always loaded
wraplist

Methods

guess_name (override)
find_perl (override)
maybe_command (override)
maybe_command_in_dirs (override)
perl_script (override)
replace_manpage_separator
init_main (override)
init_others (override)
constants (override)
cflags (override)
const_cccmd (override)
pm_to_blib (override)
tool_autosplit (override)
tool_sxubpp (override)
xsubpp_version (override)
tools_other (override)

dist (override)
c_o (override)
xs_c (override)
xs_o (override)
top_targets (override)
dlsyms (override)
dynamic_lib (override)
dynamic_bs (override)
static_lib (override)
manifypods (override)
processPL (override)
installbin (override)
subdir_x (override)
clean (override)
realclean (override)
dist_core (override)
dist_test (override)
install (override)
perldepend (override)
makefile (override)
find_tests (override)
test (override)
makeaperl (override)
nicetext (override)
prefixify (override)

ExtUtils::MM_Win32 – methods to override UN*X behaviour in

ExtUtils::MakeMaker

SYNOPSIS

DESCRIPTION

constants (o)
static_lib (o)
dynamic_bs (o)
dynamic_lib (o)
perl_script
pm_to_blib
tool_autosplit (override)
tools_other (o)

xs_o (o)
top_targets (o)
manifypods (o)
dist_ci (o)
dist_core (o)
psthru (o)

ExtUtils::MM_Win95 – method to customize MakeMaker for Win9X

SYNOPSIS
DESCRIPTION

ExtUtils::MY – ExtUtils::MakeMaker subclass for customization

SYNOPSIS
DESCRIPTION

ExtUtils::MakeMaker – create an extension Makefile

SYNOPSIS
DESCRIPTION

How To Write A Makefile.PL
Default Makefile Behaviour
make test
make testdb
make install
PREFIX and LIB attribute
AFS users
Static Linking of a new Perl Binary
Determination of Perl Library and Installation Locations
Which architecture dependent directory?
Using Attributes and Parameters

ABSTRACT, ABSTRACT_FROM, AUTHOR, BINARY_LOCATION, C, CCFLAGS, CONFIG, CONFIGURE, DEFINE, DIR, DISTNAME, DL_FUNCS, DL_VARS, EXCLUDE_EXT, EXE_FILES, FIRST_MAKEFILE, FULLPERL, FULLPERLRUN, FULLPERLRUNINST, FUNCLIST, H, IMPORTS, INC, INCLUDE_EXT, INSTALLARCHLIB, INSTALLBIN, INSTALLDIRS, INSTALLMAN1DIR, INSTALLMAN3DIR, INSTALLPRIVLIB, INSTALLSCRIPT, INSTALLSITEARCH, INSTALLSITEBIN, INSTALLSITELIB, INSTALLSITEMAN1DIR, INSTALLSITEMAN3DIR, INSTALLVENDORARCH, INSTALLVENDORBIN, INSTALLVENDORLIB, INSTALLVENDORMAN1DIR, INSTALLVENDORMAN3DIR, INST_ARCHLIB, INST_BIN, INST_LIB, INST_MAN1DIR, INST_MAN3DIR, INST_SCRIPT, LDDLFLAGS, LDFROM, LIB, LIBPERL_A, LIBS, LINKTYPE, MAKEAPERL, MAKEFILE, MAN1PODS, MAN3PODS, MAP_TARGET, MYEXTLIB, NAME, NEEDS_LINKING, NOECHO, NORECURS, NO_VC, OBJECT, OPTIMIZE, PERL, PERL_CORE, PERLMAINCC, PERL_ARCHLIB, PERL_LIB, PERL_MALLOC_OK, PERLRUN, PERLRUNINST, PERL_SRC, PERM_RW, PERM_RWX, PL_FILES, PM, PMLIBDIRS, PM_FILTER, POLLUTE, PPM_INSTALL_EXEC, PPM_INSTALL_SCRIPT, PREFIX, PREREQ_FATAL, PREREQ_PM, PREREQ_PRINT, PRINT_PREREQ, SITEPREFIX, SKIP, TYPEMAPS, VENDORPREFIX, VERBINST, VERSION, VERSION_FROM, XS, XSOPT, XSPROTOARG, XS_VERSION

Additional lowercase attributes

clean, depend, dist, dynamic_lib, linext, macro, realclean, test, tool_autosplit

Overriding MakeMaker Methods
 The End Of Cargo Cult Programming
 <MAN3PODS = ''

Hintsfile support
 Distribution Support

make distcheck, make skipcheck, make distclean, make manifest,
 make distdir, make disttest, make tardist, make dist, make
 uupload, make shdist, make zipdist, make ci

Disabling an extension
 ENVIRONMENT
 PERL_MM_OPT, PERL_MM_USE_DEFAULT

SEE ALSO
 AUTHORS

ExtUtils::Manifest – utilities to write and check a MANIFEST file

SYNOPSIS
 DESCRIPTION
 MANIFEST.SKIP
 EXPORT_OK
 GLOBAL VARIABLES
 DIAGNOSTICS

Not in MANIFEST: *file*, Skipping *file*, No such file: *file*, MANIFEST: \$!, Added to
 MANIFEST: *file*

ENVIRONMENT
 PERL_MM_MANIFEST_DEBUG

SEE ALSO
 AUTHOR

ExtUtils::Miniperl, writemain – write the C code for perlmain.c

SYNOPSIS
 DESCRIPTION
 SEE ALSO

ExtUtils::Mkbootstrap – make a bootstrap file for use by DynaLoader

SYNOPSIS
 DESCRIPTION

ExtUtils::Mksymlists – write linker options files for dynamic

extension

SYNOPSIS
 DESCRIPTION

DLBASE, DL_FUNCS, DL_VARS, FILE, FUNCLIST, IMPORTS, NAME

AUTHOR
 REVISION

ExtUtils::Packlist – manage .packlist files

SYNOPSIS
 DESCRIPTION

USAGE
FUNCTIONS

`new()`, `read()`, `write()`, `validate()`, `packlist_file()`

EXAMPLE
AUTHOR

ExtUtils::testlib – add blib/* directories to @INC

SYNOPSIS
DESCRIPTION

Fatal – replace functions with equivalents which succeed or die

SYNOPSIS
DESCRIPTION
AUTHOR

Fcntl – load the C Fcntl.h defines

SYNOPSIS
DESCRIPTION
NOTE
EXPORTED SYMBOLS

File::Basename, fileparse – split a pathname into pieces

SYNOPSIS
DESCRIPTION
`fileparse_set_fstype`, `fileparse`
EXAMPLES
`basename`, `dirname`

File::CheckTree, validate – run many filetest checks on a tree

SYNOPSIS
DESCRIPTION
AUTHOR
HISTORY

File::Compare – Compare files or filehandles

SYNOPSIS
DESCRIPTION
RETURN
AUTHOR

File::Copy – Copy files or filehandles

SYNOPSIS
DESCRIPTION
Special behaviour if `syscopy` is defined (OS/2, VMS and Win32)
`rmscopy($from, $to[, $date_flag])`
RETURN
NOTES
AUTHOR

File::DosGlob – DOS like globbing and then some

SYNOPSIS

DESCRIPTION
 NOTES
 EXPORTS (by request only)
 BUGS
 AUTHOR
 HISTORY
 SEE ALSO

File::Find – Traverse a directory tree.

SYNOPSIS
 DESCRIPTION

find, finddepth

%options

wanted, bydepth, preprocess, postprocess, follow, follow_fast,
 follow_skip, dangling_symlinks, no_chdir, untaint, untaint_pattern,
 untaint_skip

The wanted function

\$File::Find::dir is the current directory name,, \$_ is the current filename within that
 directory, \$File::Find::name is the complete pathname to the file

WARNINGS
 CAVEAT

\$dont_use_nlink, symlinks

NOTES
 HISTORY

File::Glob – Perl extension for BSD glob routine

SYNOPSIS
 DESCRIPTION

GLOB_ERR, GLOB_LIMIT, GLOB_MARK, GLOB_NOCASE, GLOB_NOCHECK, GLOB_NOSORT,
 GLOB_BRACE, GLOB_NOMAGIC, GLOB_QUOTE, GLOB_TILDE, GLOB_CSH, GLOB_ALPHASORT

DIAGNOSTICS

GLOB_NOSPACE, GLOB_ABEND

NOTES
 AUTHOR

File::Path – create or remove directory trees

SYNOPSIS
 DESCRIPTION
 AUTHORS

File::Spec – portably perform operations on file names

SYNOPSIS
 DESCRIPTION
 METHODS

canonpath, catdir, catfile, curdir, devnull, rootdir, tmpdir, updir, no_upwards, case_tolerant,
 file_name_is_absolute, path, join, splitpath, splitdir, catpath(), abs2rel, rel2abs()

SEE ALSO
 AUTHORS

File::Spec::Cygwin – methods for Cygwin file specs

SYNOPSIS
DESCRIPTION

File::Spec::Epoc – methods for Epoc file specs

SYNOPSIS
DESCRIPTION
`canonpath()`
SEE ALSO

File::Spec::Functions – portably perform operations on file names

SYNOPSIS
DESCRIPTION
Exports
SEE ALSO

File::Spec::Mac – File::Spec for Mac OS (Classic)

SYNOPSIS
DESCRIPTION
METHODS
`canonpath`
`catdir()`
`catfile`
`curdir`
`devnull`
`rootdir`
`tmpdir`
`updir`
`file_name_is_absolute`
`path`
`splitpath`
`splitdir`
`catpath`
`abs2rel`
`rel2abs`
AUTHORS
SEE ALSO

`canonpath`
`splitpath`
`splitdir`
`catpath`

File::Spec::OS2 – methods for OS/2 file specs

SYNOPSIS
DESCRIPTION

File::Spec::Unix – File::Spec for Unix, base for other File::Spec

modules

SYNOPSIS
DESCRIPTION
METHODS

`canonpath()`

`catdir()`

`catfile`

`curdir`

`devnull`

`rootdir`

`tmpdir`

`updir`

`no_upwards`

`case_tolerant`

`file_name_is_absolute`

`path`

`join`

`splitpath`

`splitdir`

`catpath()`

`abs2rel`

`rel2abs()`

SEE ALSO

File::Spec::VMS – methods for VMS file specs

SYNOPSIS
DESCRIPTION

`eliminate_macros`

`fixpath`

Methods always loaded

`canonpath (override)`

`catdir`

`catfile`

`curdir (override)`

`devnull (override)`

rootdir (override)
tmpdir (override)
updir (override)
case_tolerant (override)
path (override)
file_name_is_absolute (override)
splitpath (override)
splitdir (override)
catpath (override)
abs2rel (override)
rel2abs (override)

SEE ALSO

File::Spec::Win32 – methods for Win32 file specs

SYNOPSIS
DESCRIPTION

devnull

tmpdir
catfile
canonpath
splitpath
splitdir
catpath

Note For File::Spec::Win32 Maintainers
SEE ALSO

File::Temp – return name and handle of a temporary file safely

PORTABILITY
SYNOPSIS
DESCRIPTION
FUNCTIONS

tempfile

tempdir

MKTEMP FUNCTIONS
mkstemp

mkstemp
mkdtemp
mktemp

POSIX FUNCTIONS
tmpnam

tmpfile

ADDITIONAL FUNCTIONS

tempnam

UTILITY FUNCTIONS

unlink0

PACKAGE VARIABLES

safe_level, STANDARD, MEDIUM, HIGH

TopSystemUID

WARNING

Temporary files and NFS

HISTORY

SEE ALSO

AUTHOR

File::stat – by-name interface to Perl's built-in stat() functions

SYNOPSIS

DESCRIPTION

BUGS

NOTE

AUTHOR

FileCache – keep more files open than the system permits

SYNOPSIS

DESCRIPTION

cacheout EXPR, cacheout MODE, EXPR

CAVEATS

BUGS

FileHandle – supply object methods for filehandles

SYNOPSIS

DESCRIPTION

\$fh-print, \$fh-printf, \$fh-getline, \$fh-getlines

SEE ALSO

Filter::Simple – Simplified source filtering

SYNOPSIS

DESCRIPTION

The Problem

A Solution

Disabling or changing <no behaviour

All-in-one interface

Filtering only specific components of source code

`"code", "executable", "quotelike", "string", "regex", "all"`

Filtering only the code parts of source code

Most source code ceases to be grammatically correct when it is broken up into the pieces between string literals and regexes. So the 'code' component filter behaves slightly differently from the other partial filters described in the previous section.

Using Filter::Simple with an explicit `import` subroutine
Using Filter::Simple and Exporter together
How it works
AUTHOR
COPYRIGHT

Filter::Util::Call – Perl Source Filter Utility Module

SYNOPSIS
DESCRIPTION
 use Filter::Util::Call
 import()
 filter() and anonymous sub
 \$_, **\$status**, **filter_read** and **filter_read_exact**, **filter_del**
EXAMPLES
 Example 1: A simple filter.
 Example 2: Using the context
 Example 3: Using the context within the filter
 Example 4: Using `filter_del`
Filter::Simple
AUTHOR
DATE

FindBin – Locate directory of original perl script

SYNOPSIS
DESCRIPTION
EXPORTABLE VARIABLES
KNOWN ISSUES
KNOWN BUGS
AUTHORS
COPYRIGHT

GDBM_File – Perl5 access to the gdbm library.

SYNOPSIS
DESCRIPTION
AVAILABILITY
BUGS
SEE ALSO

Getopt::Long – Extended processing of command line options

SYNOPSIS
DESCRIPTION
Command Line Options, an Introduction
Getting Started with Getopt::Long
 Simple options
 A little bit less simple options
 Mixing command line option with other arguments
 Options with values
 Options with multiple values
 Options with hash values
 User-defined subroutines to handle options
 Options with multiple names
 Case and abbreviations

Summary of Option Specifications

!, +, s, i, o, f, : *type* [*desttype*], : *number* [*desttype*], : + [*desttype*]

Advanced Possibilities

- Object oriented interface
- Thread Safety
- Documentation and help texts
- Storing options in a hash
- Bundling
- The lonesome dash
- Argument callback

Configuring Getopt::Long

default, posix_default, auto_abbrev, getopt_compat, gnu_compat, gnu_getopt, require_order, permute, bundling (default: disabled), bundling_override (default: disabled), ignore_case (default: enabled), ignore_case_always (default: disabled), pass_through (default: disabled), prefix, prefix_pattern, debug (default: disabled)

Return values and Errors

Legacy

- Default destinations
- Alternative option starters
- Configuration variables

Trouble Shooting

- Warning: Ignoring '!' modifier for short option
- GetOptions does not return a false result when an option is not supplied
- GetOptions does not split the command line correctly
- How do I put a "-?" option into a Getopt::Long?

AUTHOR

COPYRIGHT AND DISCLAIMER

Getopt::Std, getopt – Process single-character switches with switch

clustering

SYNOPSIS

DESCRIPTION

Hash::Util – A selection of general-utility hash subroutines

SYNOPSIS

DESCRIPTION

Restricted hashes

lock_keys, unlock_keys

lock_value, unlock_value

lock_hash, unlock_hash

AUTHOR

SEE ALSO

I18N::Collate – compare 8-bit scalar data according to the current

locale

SYNOPSIS

DESCRIPTION

I18N::LangTags – functions for dealing with RFC3066–style language

tags

SYNOPSIS

DESCRIPTION

the function `is_language_tag($lang1)`the function `extract_language_tags($whatever)`the function `same_language_tag($lang1, $lang2)`the function `similarity_language_tag($lang1, $lang2)`the function `is_dialect_of($lang1, $lang2)`the function `super_languages($lang1)`the function `locale2language_tag($locale_identifier)`the function `encode_language_tag($lang1)`the function `alternate_language_tags($lang1)`the function `@langs = panic_languages(@accept_languages)`

ABOUT LOWERCASING

ABOUT UNICODE PLAINTEXT LANGUAGE TAGS

SEE ALSO

COPYRIGHT

AUTHOR

I18N::LangTags::List — tags and names for human languages

SYNOPSIS

DESCRIPTION

ABOUT LANGUAGE TAGS

LIST OF LANGUAGES

{ab} : Abkhazian, {ace} : Achinese, {ach} : Acoli, {ada} : Adangme, {aa} : Afar, {afh} : Afrihili, {af} : Afrikaans, [{afa} : Afro–Asiatic (Other)], {aka} : Akan, {akk} : Akkadian, {sq} : Albanian, {ale} : Aleut, [{alg} : Algonquian languages], [{tut} : Altaic (Other)], {am} : Amharic, {i-ami} : Ami, [{apa} : Apache languages], {ar} : Arabic, {arc} : Aramaic, {arp} : Arapaho, {arn} : Araucanian, {arw} : Arawak, {hy} : Armenian, [{art} : Artificial (Other)], {as} : Assamese, [{ath} : Athapaskan languages], [{aus} : Australian languages], [{map} : Austronesian (Other)], {ava} : Avaric, {ae} : Avestan, {awa} : Awadhi, {ay} : Aymara, {az} : Azerbaijani, {ban} : Balinese, [{bat} : Baltic (Other)], {bal} : Baluchi, {bam} : Bambara, [{bai} : Bamileke languages], {bad} : Banda, [{bnt} : Bantu (Other)], {bas} : Basa, {ba} : Bashkir, {eu} : Basque, {btk} : Batak (Indonesia), {bej} : Beja, {be} : Belarusian, {bem} : Bemba, {bn} : Bengali, [{ber} : Berber (Other)], {bho} : Bhojpuri, {bh} : Bihari, {bik} : Bikol, {bin} : Bini, {bi} : Bislama, {bs} : Bosnian, {bra} : Braj, {br} : Breton, {bug} : Buginese, {bg} : Bulgarian, {i-bnn} : Bunun, {bua} : Buriat, {my} : Burmese, {cad} : Caddo, {car} : Carib, {ca} : Catalan, [{cau} : Caucasian (Other)], {ceb} : Cebuano, [{cel} : Celtic (Other)], [{cai} : Central American Indian (Other)], {chg} : Chagatai, [{cmc} : Chamic languages], {ch} : Chamorro, {ce} : Chechen, {chr} : Cherokee, {chy} : Cheyenne, {chb} : Chibcha, {ny} : Chichewa, {zh} : Chinese, {chn} : Chinook Jargon, {chp} : Chipewyan, {cho} : Choctaw, {cu} : Church Slavic, {chk} : Chuukese, {cv} : Chuvash, {cop} : Coptic, {kw} : Cornish, {co} : Corsican, {cre} : Cree, {mus} : Creek, [{cpe} : English–based Creoles and pidgins (Other)], [{cpf} : French–based Creoles and pidgins (Other)], [{cpp} : Portuguese–based Creoles and pidgins (Other)], [{crp} : Creoles and pidgins (Other)], {hr} : Croatian, [{cus} : Cushitic (Other)], {cs} : Czech, {dak} : Dakota, {da} : Danish, {day} : Dayak, {i-default} : Default (Fallthru) Language, {del} : Delaware, {din} : Dinka, {div} : Divehi, {doi} : Dogri, {dgr} : Dogrib, [{dra} : Dravidian (Other)], {dua} : Duala, {nl} : Dutch, {dum}

: Middle Dutch (ca.1050–1350), {dyu} : Dyula, {dz} : Dzongkha, {efi} : Efik, {egy} : Ancient Egyptian, {eka} : Ekajuk, {elx} : Elamite, {en} : English, {enm} : Old English (1100–1500), {ang} : Old English (ca.450–1100), {eo} : Esperanto, {et} : Estonian, {ewe} : Ewe, {ewo} : Ewondo, {fan} : Fang, {fat} : Fanti, {fo} : Faroese, {fj} : Fijian, {fi} : Finnish, [{fiu} : Finno–Ugrian (Other)], {fon} : Fon, {fr} : French, {frm} : Middle French (ca.1400–1600), {fro} : Old French (842–ca.1400), {fy} : Frisian, {fur} : Friulian, {ful} : Fulah, {gaa} : Ga, {gd} : Scots Gaelic, {gl} : Gallegan, {lug} : Ganda, {gay} : Gayo, {gba} : Gbaya, {gez} : Geez, {ka} : Georgian, {de} : German, {gmh} : Middle High German (ca.1050–1500), {goh} : Old High German (ca.750–1050), [{gem} : Germanic (Other)], {gil} : Gilbertese, {gon} : Gondi, {gor} : Gorontalo, {got} : Gothic, {grb} : Grebo, {grc} : Ancient Greek, {el} : Modern Greek, {gn} : Guarani, {gu} : Gujarati, {gwi} : Gwich'in, {hai} : Haida, {ha} : Hausa, {haw} : Hawaiian, {he} : Hebrew, {hz} : Herero, {hil} : Hiligaynon, {him} : Himachali, {hi} : Hindi, {ho} : Hiri Motu, {hit} : Hittite, {hmn} : Hmong, {hu} : Hungarian, {hup} : Hupa, {iba} : Iban, {is} : Icelandic, {ibo} : Igbo, {ijo} : Ijo, {ilo} : Iloko, [{inc} : Indic (Other)], [{ine} : Indo–European (Other)], {id} : Indonesian, {ia} : Interlingua (International Auxiliary Language Association), {ie} : Interlingue, {iu} : Inuktitut, {ik} : Inupiaq, [{ira} : Iranian (Other)], {ga} : Irish, {mga} : Middle Irish (900–1200), {sga} : Old Irish (to 900), [{iro} : Iroquoian languages], {it} : Italian, {ja} : Japanese, {jw} : Javanese, {jrb} : Judeo–Arabic, {jpr} : Judeo–Persian, {kab} : Kabyle, {kac} : Kachin, {kl} : Kalaallisut, {kam} : Kamba, {kn} : Kannada, {kau} : Kanuri, {kaa} : Kara–Kalpak, {kar} : Karen, {ks} : Kashmiri, {kaw} : Kawi, {kk} : Kazakh, {kha} : Khasi, {km} : Khmer, [{khi} : Khoisan (Other)], {kho} : Khotanese, {ki} : Kikuyu, {kmb} : Kimbundu, {rw} : Kinyarwanda, {ky} : Kirghiz, {i-klingon} : Klingon, {kv} : Komi, {kon} : Kongo, {kok} : Konkani, {ko} : Korean, {kos} : Kosraean, {kpe} : Kpelle, {kro} : Kru, {kj} : Kuanyama, {kum} : Kumyk, {ku} : Kurdish, {kru} : Kurukh, {kut} : Kutenai, {lad} : Ladino, {lah} : Lahnda, {lam} : Lamba, {lo} : Lao, {la} : Latin, {lv} : Latvian, {lb} : Letzeburgesch, {lez} : Lezghian, {ln} : Lingala, {lt} : Lithuanian, {nds} : Low German, {loz} : Lozi, {lub} : Luba–Katanga, {lua} : Luba–Lulua, {lui} : Luiseno, {lun} : Lunda, {luo} : Luo (Kenya and Tanzania), {lus} : Lushai, {mk} : Macedonian, {mad} : Madurese, {mag} : Magahi, {mai} : Maithili, {mak} : Makasar, {mg} : Malagasy, {ms} : Malay, {ml} : Malayalam, {mt} : Maltese, {mnc} : Manchu, {mdr} : Mandar, {man} : Mandingo, {mni} : Manipuri, [{mno} : Manobo languages], {gv} : Manx, {mi} : Maori, {mr} : Marathi, {chm} : Mari, {mh} : Marshall, {mwr} : Marwari, {mas} : Masai, [{myn} : Mayan languages], {men} : Mende, {mic} : Micmac, {min} : Minangkabau, {i-mingo} : Mingo, [{mis} : Miscellaneous languages], {moh} : Mohawk, {mo} : Moldavian, [{mkh} : Mon–Khmer (Other)], {lol} : Mongo, {mn} : Mongolian, {mos} : Mossi, [{mul} : Multiple languages], [{mun} : Munda languages], {nah} : Nahuatl, {na} : Nauru, {nv} : Navajo, {nd} : North Ndebele, {nr} : South Ndebele, {ng} : Ndonga, {ne} : Nepali, {new} : Newari, {nia} : Nias, [{nic} : Niger–Kordofanian (Other)], [{ssa} : Nilo–Saharan (Other)], {niu} : Niuean, {non} : Old Norse, [{nai} : North American Indian], {se} : Northern Sami, {no} : Norwegian, {nb} : Norwegian Bokmal, {nn} : Norwegian Nynorsk, [{nub} : Nubian languages], {nym} : Nyamwezi, {nyn} : Nyankole, {nyo} : Nyoro, {nzi} : Nzima, {oc} : Occitan (post 1500), {oji} : Ojibwa, {or} : Oriya, {om} : Oromo, {osa} : Osage, {os} : Ossetian; Ossetic, [{oto} : Otomian languages], {pal} : Pahlavi, {i-pwn} : Paiwan, {pau} : Palauan, {pi} : Pali, {pam} : Pampanga, {pag} : Pangasinan, {pa} : Panjabi, {pap} : Papiamentu, [{paa} : Papuan (Other)], {fa} : Persian, {peo} : Old Persian (ca.600–400 B.C.), [{phi} : Philippine (Other)], {phn} : Phoenician, {pon} : Pohnpeian, {pl} : Polish, {pt} : Portuguese, [{pra} : Prakrit languages], {pro} : Old Provencal (to 1500), {ps} : Pushto, {qu} : Quechua, {rm} : Raeto–Romance, {raj} : Rajasthani, {rap} : Rapanui, {rar} : Rarotongan, [{qaa – qtz} : Reserved for local use.], [{roa} : Romance (Other)], {ro} : Romanian, {rom} : Romany, {rn} : Rundi, {ru} : Russian, [{sal} : Salishan languages], {sam} : Samaritan Aramaic, [{smi} : Sami languages (Other)], {sm} : Samoan, {sad} : Sandawe, {sg} : Sango, {sa} : Sanskrit, {sat} : Santali, {sc} : Sardinian, {sas} : Sasak, {sco} : Scots, {sel} : Selkup, [{sem} : Semitic (Other)], {sr} : Serbian, {srr} : Serer, {shn} : Shan, {sn} : Shona, {sid} : Sidamo, {sgn-...} : Sign Languages, {bla} : Siksika, {sd} : Sindhi, {si} : Sinhalese, [{sit} : Sino–Tibetan (Other)], [{sio} : Siouan languages], {den} : Slave (Athapaskan), [{sla} : Slavic (Other)], {sk} : Slovak, {sl} : Slovenian, {sog} : Sogdian, {so} : Somali, {son} : Songhai, {snk} : Soninke, {wen} : Sorbian languages, {nso} : Northern Sotho, {st} : Southern Sotho, [{sai} : South American Indian (Other)], {es} : Spanish, {suk} : Sukuma, {sux} : Sumerian, {su} : Sundanese, {sus} : Susu, {sw} : Swahili, {ss} : Swati, {sv} : Swedish, {syr} : Syriac,

{tl} : Tagalog, {ty} : Tahitian, [{tai} : Tai (Other)], {tg} : Tajik, {tmh} : Tamashek, {ta} : Tamil, {i-tao} : Tao, {tt} : Tatar, {i-tay} : Tayal, {te} : Telugu, {ter} : Tereno, {tet} : Tetum, {th} : Thai, {bo} : Tibetan, {tig} : Tigre, {ti} : Tigrinya, {tem} : Timne, {tiv} : Tiv, {tli} : Tlingit, {tpi} : Tok Pisin, {tkl} : Tokelau, {tog} : Tonga (Nyasa), {to} : Tonga (Tonga Islands), {tsi} : Tsimshian, {ts} : Tsonga, {i-tsu} : Tsou, {tn} : Tswana, {tum} : Tumbuka, {tr} : Turkish, {ota} : Ottoman Turkish (1500–1928), {tk} : Turkmen, {tvl} : Tuvalu, {tyv} : Tuvinian, {tw} : Twi, {uga} : Ugaritic, {ug} : Uighur, {uk} : Ukrainian, {umb} : Umbundu, {und} : Undetermined, {ur} : Urdu, {uz} : Uzbek, {vai} : Vai, {ven} : Venda, {vi} : Vietnamese, {vo} : Volapuk, {vot} : Votic, [{wak} : Wakashan languages], {wal} : Walamo, {war} : Waray, {was} : Washo, {cy} : Welsh, {wo} : Wolof, {x-...} : Unregistered (Semi–Private Use), {xh} : Xhosa, {sah} : Yakut, {yao} : Yao, {yap} : Yapese, {yi} : Yiddish, {yo} : Yoruba, [{ypk} : Yupik languages], {znd} : Zande, [{zap} : Zapotec], {zen} : Zenaga, {za} : Zhuang, {zu} : Zulu, {zun} : Zuni

SEE ALSO

COPYRIGHT AND DISCLAIMER

AUTHOR

I18N::Langinfo – query locale information

SYNOPSIS

DESCRIPTION

EXPORT

SEE ALSO

AUTHOR

COPYRIGHT AND LICENSE

IO – load various IO modules

SYNOPSIS

DESCRIPTION

IO::Dir – supply object methods for directory handles

SYNOPSIS

DESCRIPTION

```
new ( [ DIRNAME ] ), open ( DIRNAME ), read ( ), seek ( POS ), tell ( ), rewind ( ), close ( ), tie
%hash, IO::Dir, DIRNAME [, OPTIONS ]
```

SEE ALSO

AUTHOR

COPYRIGHT

IO::File – supply object methods for filehandles

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

```
new ( FILENAME [,MODE [,PERMS]] ), new_tmpfile
```

METHODS

```
open( FILENAME [,MODE [,PERMS]] )
```

SEE ALSO

HISTORY

IO::Handle – supply object methods for I/O handles

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

```
new ( ), new_from_fd ( FD, MODE )
```

METHODS

```
$io->fdopen ( FD, MODE ), $io->opened, $io->getline, $io->getlines, $io->ungetc ( ORD ),
$io->write ( BUF, LEN [, OFFSET ] ), $io->error, $io->clearerr, $io->sync, $io->flush,
$io->printflush ( ARGS ), $io->blocking ( [ BOOL ] ), $io->untaint
```

NOTE**SEE ALSO****BUGS****HISTORY****IO::Pipe – supply object methods for pipes****SYNOPSIS****DESCRIPTION****CONSTRUCTOR**

```
new ( [READER, WRITER] )
```

METHODS

```
reader ([ARGS]), writer ([ARGS]), handles ( )
```

SEE ALSO**AUTHOR****COPYRIGHT****IO::Poll – Object interface to system poll call****SYNOPSIS****DESCRIPTION****METHODS**

```
mask ( IO [, EVENT_MASK ] ), poll ( [ TIMEOUT ] ), events ( IO ), remove ( IO ), handles ( [
EVENT_MASK ] )
```

SEE ALSO**AUTHOR****COPYRIGHT****IO::Seekable – supply seek based methods for I/O objects****SYNOPSIS****DESCRIPTION**

```
$io->getpos, $io->setpos, $io->seek ( POS, WHENCE ), WHENCE=0 (SEEK_SET), WHENCE=1
(SEEK_CUR), WHENCE=2 (SEEK_END), $io->sysseek( POS, WHENCE ), $io->tell
```

SEE ALSO**HISTORY****IO::Select – OO interface to the select system call****SYNOPSIS****DESCRIPTION****CONSTRUCTOR**

```
new ( [ HANDLES ] )
```

METHODS

```
add ( HANDLES ), remove ( HANDLES ), exists ( HANDLE ), handles, can_read ( [ TIMEOUT ] ),
can_write ( [ TIMEOUT ] ), has_exception ( [ TIMEOUT ] ), count ( ), bits ( ), select ( READ,
WRITE, ERROR [, TIMEOUT ] )
```


EXAMPLE
AUTHOR
COPYRIGHT

IO::Socket – Object interface to socket communications

SYNOPSIS
DESCRIPTION
CONSTRUCTOR

new ([ARGS])

METHODS

accept([PKG]), socketpair(DOMAIN, TYPE, PROTOCOL), atmark, connected, protocol, sockdomain,
sockopt(OPT [, VAL]), socktype, timeout([VAL])

SEE ALSO
AUTHOR
COPYRIGHT

IO::Socket::INET – Object interface for AF_INET domain sockets

SYNOPSIS
DESCRIPTION
CONSTRUCTOR

new ([ARGS])

METHODS

sockaddr (), sockport (), sockhost (), peeraddr (), peerport (), peerhost ()

SEE ALSO
AUTHOR
COPYRIGHT

IO::Socket::UNIX – Object interface for AF_UNIX domain sockets

SYNOPSIS
DESCRIPTION
CONSTRUCTOR

new ([ARGS])

METHODS

hostpath (), peerpath ()

SEE ALSO
AUTHOR
COPYRIGHT

IO::lib::IO::Dir, IO::Dir – supply object methods for directory

handles

SYNOPSIS
DESCRIPTION

new ([DIRNAME]), open (DIRNAME), read (), seek (POS), tell (), rewind (), close (), tie
%hash, IO::Dir, DIRNAME [, OPTIONS]

SEE ALSO
AUTHOR
COPYRIGHT

IO::lib::IO::File, IO::File – supply object methods for filehandles

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

`new (FILENAME [,MODE [,PERMS]]), new_tmpfile`

METHODS

`open(FILENAME [,MODE [,PERMS]])`

SEE ALSO

HISTORY

IO::lib::IO::Handle, IO::Handle – supply object methods for I/O

handles

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

`new (), new_from_fd (FD, MODE)`

METHODS

`$io-fdopen (FD, MODE), $io-opened, $io-getline, $io-getlines, $io-ungetc (ORD),
$io-write (BUF, LEN [, OFFSET]), $io-error, $io-clearerr, $io-sync, $io-flush,
$io-printflush (ARGS), $io-blocking ([BOOL]), $io-untaint`

NOTE

SEE ALSO

BUGS

HISTORY

IO::lib::IO::Pipe, IO::Pipe – supply object methods for pipes

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

`new ([READER, WRITER])`

METHODS

`reader ([ARGS]), writer ([ARGS]), handles ()`

SEE ALSO

AUTHOR

COPYRIGHT

IO::lib::IO::Poll, IO::Poll – Object interface to system poll call

SYNOPSIS

DESCRIPTION

METHODS

`mask (IO [, EVENT_MASK]), poll ([TIMEOUT]), events (IO), remove (IO), handles([
EVENT_MASK])`

SEE ALSO

AUTHOR

COPYRIGHT

IO::lib::IO::Seekable, IO::Seekable – supply seek based methods for

I/O objects

SYNOPSIS

DESCRIPTION

`$io-getpos, $io-setpos, $io-seek (POS, WHENCE), WHENCE=0 (SEEK_SET), WHENCE=1 (SEEK_CUR), WHENCE=2 (SEEK_END), $io-sysseek(POS, WHENCE), $io-tell`

SEE ALSO

HISTORY

IO::lib::IO::Select, IO::Select – OO interface to the select system

call

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

`new ([HANDLES])`

METHODS

`add (HANDLES), remove (HANDLES), exists (HANDLE), handles, can_read ([TIMEOUT]), can_write ([TIMEOUT]), has_exception ([TIMEOUT]), count (), bits (), select (READ, WRITE, ERROR [, TIMEOUT])`

EXAMPLE

AUTHOR

COPYRIGHT

IO::lib::IO::Socket, IO::Socket – Object interface to socket

communications

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

`new ([ARGS])`

METHODS

`accept([PKG]), socketpair(DOMAIN, TYPE, PROTOCOL), atmark, connected, protocol, sockdomain, sockopt(OPT [, VAL]), socktype, timeout([VAL])`

SEE ALSO

AUTHOR

COPYRIGHT

IO::lib::IO::Socket::INET, IO::Socket::INET – Object interface for

AF_INET domain sockets

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

`new ([ARGS])`

METHODS

`sockaddr (), sockport (), sockhost (), peeraddr (), peerport (), peerhost ()`

SEE ALSO

AUTHOR

COPYRIGHT

IO::lib::IO::Socket::UNIX, IO::Socket::UNIX – Object interface for

AF_UNIX domain sockets

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

new ([ARGS])

METHODS

hostpath(), peerpath()

SEE ALSO

AUTHOR

COPYRIGHT

IPC::Msg – SysV Msg IPC object class

SYNOPSIS

DESCRIPTION

METHODS

new (KEY , FLAGS), id, rcv (BUF, LEN [, TYPE [, FLAGS]]), remove, set (STAT), set (NAME
= VALUE [, NAME = VALUE ...]), snd (TYPE, MSG [, FLAGS]), stat

SEE ALSO

AUTHOR

COPYRIGHT

IPC::Open2, open2 – open a process for both reading and writing

SYNOPSIS

DESCRIPTION

WARNING

SEE ALSO

IPC::Open3, open3 – open a process for reading, writing, and error

handling

SYNOPSIS

DESCRIPTION

WARNING

IPC::Semaphore – SysV Semaphore IPC object class

SYNOPSIS

DESCRIPTION

METHODS

new (KEY , NSEMS , FLAGS), getall, getncnt (SEM), getpid (SEM), getval (SEM), getzcnt (SEM), id, op (OPLIST), remove, set (STAT), set (NAME = VALUE [, NAME = VALUE ...]),
setall (VALUES), setval (N , VALUE), stat

SEE ALSO

AUTHOR

COPYRIGHT

IPC::SysV – SysV IPC constants

SYNOPSIS

DESCRIPTION

ftok(PATH, ID)

SEE ALSO
AUTHORS
COPYRIGHT

IPC::SysV::Msg, IPC::Msg – SysV Msg IPC object class

SYNOPSIS
DESCRIPTION
METHODS

new (KEY , FLAGS), id, rcv (BUF, LEN [, TYPE [, FLAGS]]), remove, set (STAT), set (NAME = VALUE [, NAME = VALUE ...]), snd (TYPE, MSG [, FLAGS]), stat

SEE ALSO
AUTHOR
COPYRIGHT

IPC::SysV::Semaphore, IPC::Semaphore – SysV Semaphore IPC object

class

SYNOPSIS
DESCRIPTION
METHODS

new (KEY , NSEMS , FLAGS), getall, getncnt (SEM), getpid (SEM), getval (SEM), getzcnt (SEM), id, op (OPLIST), remove, set (STAT), set (NAME = VALUE [, NAME = VALUE ...]), setall (VALUES), setval (N , VALUE), stat

SEE ALSO
AUTHOR
COPYRIGHT

List::Util – A selection of general–utility list subroutines

SYNOPSIS
DESCRIPTION

first BLOCK LIST, max LIST, maxstr LIST, min LIST, minstr LIST, reduce BLOCK LIST, shuffle LIST, sum LIST

KNOWN BUGS
SUGGESTED ADDITIONS
COPYRIGHT

List::Utilib::List::Util, List::Util – A selection of

general–utility list subroutines

SYNOPSIS
DESCRIPTION

first BLOCK LIST, max LIST, maxstr LIST, min LIST, minstr LIST, reduce BLOCK LIST, shuffle LIST, sum LIST

KNOWN BUGS
SUGGESTED ADDITIONS
COPYRIGHT

List::Utilib::Scalar::Util, Scalar::Util – A selection of

general–utility scalar subroutines

SYNOPSIS
DESCRIPTION

blessed EXPR, dualvar NUM, STRING, isweak EXPR, openhandle FH, reftype EXPR, tainted EXPR,

weaken REF

KNOWN BUGS

COPYRIGHT

BLATANT PLUG

Locale::Constants – constants for Locale codes

SYNOPSIS

DESCRIPTION

KNOWN BUGS AND LIMITATIONS

SEE ALSO

Locale::Language, Locale::Country, Locale::Script, Locale::Currency

AUTHOR

COPYRIGHT

Locale::Country – ISO codes for country identification (ISO 3166)

SYNOPSIS

DESCRIPTION

alpha-2, alpha-3, numeric

CONVERSION ROUTINES

code2country(CODE, [CODESET]), country2code(STRING, [CODESET]), country_code2code(CODE, CODESET, CODESET)

QUERY ROUTINES

all_country_codes([CODESET]), all_country_names([CODESET])

SEMI-PRIVATE ROUTINES

alias_code

rename_country

EXAMPLES

DOMAIN NAMES

KNOWN BUGS AND LIMITATIONS

SEE ALSO

Locale::Language, Locale::Script, Locale::Currency, Locale::SubCountry, ISO 3166-1,
<http://www.iso.org/iso/en/prods-services/iso3166ma/index.html>,
<http://www.egt.ie/standards/iso3166/iso3166-1-en.html>,
<http://www.cia.gov/cia/publications/factbook/docs/app-d-1.html>

AUTHOR

COPYRIGHT

Locale::Currency – ISO three letter codes for currency

identification (ISO 4217)

SYNOPSIS

DESCRIPTION

XTS, XXX

CONVERSION ROUTINES

code2currency(), currency2code()

QUERY ROUTINES

all_currency_codes(), all_currency_names()

EXAMPLES

KNOWN BUGS AND LIMITATIONS

SEE ALSO

Locale::Country, Locale::Script, ISO 4217:1995, <http://www.bsi-global.com/iso4217currency>

AUTHOR

COPYRIGHT

Locale::Language – ISO two letter codes for language identification

(ISO 639)

SYNOPSIS

DESCRIPTION

CONVERSION ROUTINES

`code2language()`, `language2code()`

QUERY ROUTINES

`all_language_codes()`, `all_language_names()`

EXAMPLES

KNOWN BUGS AND LIMITATIONS

SEE ALSO

Locale::Country, Locale::Script, Locale::Currency, ISO 639:1988 (E/F),
<http://lcweb.loc.gov/standards/iso639-2/langhome.html>

AUTHOR

COPYRIGHT

Locale::Maketext — framework for localization

SYNOPSIS

DESCRIPTION

QUICK OVERVIEW

METHODS

Construction Methods

The "maketext" Method

`$lh-fail_with` or `$lh-fail_with(PARAM)`, `$lh-failure_handler_auto`

Utility Methods

`$language-quant($number, $singular)`, `$language-quant($number, $singular, $plural)`,
`$language-quant($number, $singular, $plural, $negative)`,
`$language-numf($number)`, `$language-sprintf($format, @items)`,
`$language-language_tag()`, `$language-encoding()`

Language Handle Attributes and Internals

LANGUAGE CLASS HIERARCHIES

ENTRIES IN EACH LEXICON

BRACKET NOTATION

AUTO LEXICONS

CONTROLLING LOOKUP FAILURE

HOW TO USE MAKETEXT

SEE ALSO

COPYRIGHT AND DISCLAIMER

AUTHOR

Locale::Maketext::TPJ13 — article about software localization

SYNOPSIS

DESCRIPTION

Localization and Perl: gettext breaks, Maketext fixes

A Localization Horror Story: It Could Happen To You

The Linguistic View

Breaking gettext

Replacing gettext

Buzzwords: Abstraction and Encapsulation

Buzzword: Isomorphism

Buzzword: Inheritance

Buzzword: Concision

The Devil in the Details

The Proof in the Pudding: Localizing Web Sites

References

Locale::Script – ISO codes for script identification (ISO 15924)

SYNOPSIS

DESCRIPTION

alpha-2, alpha-3, numeric

SPECIAL CODES

CONVERSION ROUTINES

code2script(CODE, [CODESET]), script2code(STRING, [CODESET]), script_code2code(CODE, CODESET, CODESET)

QUERY ROUTINES

all_script_codes ([CODESET]), all_script_names ([CODESET])

EXAMPLES

KNOWN BUGS AND LIMITATIONS

SEE ALSO

Locale::Language, Locale::Currency, Locale::Country, ISO 15924,
<http://www.evertype.com/standards/iso15924/>

AUTHOR

COPYRIGHT

MIME::Base64 – Encoding and decoding of base64 strings

SYNOPSIS

DESCRIPTION

encode_base64(\$str, [\$eol]), decode_base64(\$str)

DIAGNOSTICS

Premature end of base64 data, Premature padding of base64 data

EXAMPLES

COPYRIGHT

MIME::Base64::QuotedPrint, MIME::QuotedPrint – Encoding and decoding

of quoted-printable strings

SYNOPSIS

DESCRIPTION

encode_qp(\$str), decode_qp(\$str);

COPYRIGHT

MIME::QuotedPrint – Encoding and decoding of quoted–printable

strings

SYNOPSIS

DESCRIPTION

```
encode_qp($str) , decode_qp($str) ;
```

COPYRIGHT

Math::BigFloat – Arbitrary size floating point math package

SYNOPSIS

DESCRIPTION

Canonical notation

Output

```
mantissa() , exponent() and parts()
```

Accuracy vs. Precision

Rounding

```
ffround ( +$scale ) , fround ( -$scale ) , fround ( 0 ) , fround ( +$scale ) , fround (
-$scale ) and fround ( 0 )
```

EXAMPLES

```
# not ready yet
```

Autocreating constants

Math library

Using Math::BigInt::Lite

BUGS

CAVEAT

```
stringify, bstr() , bdiv, Modifying and =, bpow
```

LICENSE

AUTHORS

Math::BigInt – Arbitrary size integer math package

SYNOPSIS

DESCRIPTION

Canonical notation, Input, Output

METHODS

config

accuracy

brsft

new

bnan

bzero

binf

bone

```
is_one() / is_zero() / is_nan() / is_inf()
```

```
is_positive() / is_negative()
```

```
$x->is_positive();           # true if >= 0
```

```
$x->is_negative();          # true if < 0
```

`is_odd()/is_even()/is_int()`
`bcmp`
`bacmp`
`sign`
`bcmp`
`bneg`
`babs`
`bnorm`
`bnot`
`binc`
`bdec`
`badd`
`bsub`
`bmul`
`bdiv`
`bmod`
`bmodinv`
`bmodpow`
`bpow`
`blsft`
`brsft`
`band`
`bior`
`bxor`
`bnot`
`bsqrt`
`bfac`
`round`
`bround`
`bfround`
`bfloor`
`bceil`
`bgcd`
`blcm`
`exponent`
`mantissa`
`parts`
`copy`
`as_number`
`bsstr`
`as_hex`
`as_bin`
ACCURACY and PRECISION
Precision P
Accuracy A
Fallback F
Rounding mode R
 ‘trunc’, ‘even’, ‘odd’, ‘+inf’, ‘-inf’, ‘zero’, Precision, Accuracy (significant digits),
 Setting/Accessing, Creating numbers, Usage, Precedence, Overriding globals, Local settings,
 Rounding, Default values, Remarks
INTERNALS
MATH LIBRARY

SIGN
 mantissa(), exponent() and parts()
 EXAMPLES
 use Math::BigInt;
 Autocreating constants
 PERFORMANCE
 Alternative math libraries
 SUBCLASSING
 Subclassing Math::BigInt
 UPGRADING
 Auto-upgrade
 bsqrt(), div(), blog()
 BUGS
 Out of Memory!, Fails to load Calc on Perl prior 5.6.0
 CAVEATS
 stringify, bstr(), bsstr() and 'cmp', int(), length, bdiv, infinity handling, Modifying and =,
 bpow, Overloading - \$x, Mixing different object types, bsqrt(), brsft()
 LICENSE
 SEE ALSO
 AUTHORS

Math::BigInt::Calc – Pure Perl module to support Math::BigInt

SYNOPSIS
 DESCRIPTION
 EXPORT
 WRAP YOUR OWN
 LICENSE
 This program is free software; you may redistribute it and/or modify it under the same terms as Perl
 itself.
 AUTHORS
 SEE ALSO

Math::BigRat – arbitrarily big rationals

SYNOPSIS
 DESCRIPTION
 MATH LIBRARY
 METHODS
 new()
 numerator()
 denominator()
 \$d = \$x->denominator();
 parts()
 as_number()
 bfac()
 blog()
 bround()/round()/bfround()
 BUGS
 LICENSE
 SEE ALSO

AUTHORS

Math::Complex – complex numbers and associated mathematical

functions

SYNOPSIS

DESCRIPTION

OPERATIONS

CREATION

STRINGIFICATION

CHANGED IN PERL 5.6

USAGE

ERRORS DUE TO DIVISION BY ZERO OR LOGARITHM OF ZERO

ERRORS DUE TO INDIGESTIBLE ARGUMENTS

BUGS

AUTHORS

Math::Trig – trigonometric functions

SYNOPSIS

DESCRIPTION

TRIGONOMETRIC FUNCTIONS

tan

ERRORS DUE TO DIVISION BY ZERO

SIMPLE (REAL) ARGUMENTS, COMPLEX RESULTS

PLANE ANGLE CONVERSIONS

RADIAL COORDINATE CONVERSIONS

COORDINATE SYSTEMS

3-D ANGLE CONVERSIONS

cartesian_to_cylindrical, cartesian_to_spherical, cylindrical_to_cartesian,
cylindrical_to_spherical, spherical_to_cartesian, spherical_to_cylindrical

GREAT CIRCLE DISTANCES AND DIRECTIONS

EXAMPLES

CAVEAT FOR GREAT CIRCLE FORMULAS

BUGS

AUTHORS

Memoize – Make functions faster by trading space for time

SYNOPSIS

DESCRIPTION

DETAILS

OPTIONS

INSTALL

NORMALIZER

SCALAR_CACHE, LIST_CACHE

MEMORY, HASH, TIE, FAULT, MERGE

OTHER FACILITIES

unmemoize

flush_cache

CAVEATS

PERSISTENT CACHE SUPPORT

EXPIRATION SUPPORT

BUGS

MAILING LIST
AUTHOR
COPYRIGHT AND LICENSE
THANK YOU

Memoize::AnyDBM_File – glue to provide EXISTS for AnyDBM_File for

Storable use

DESCRIPTION

Memoize::Expire – Plug-in module for automatic expiration of

memoized values

SYNOPSIS
DESCRIPTION
INTERFACE

TIEHASH, EXISTS, STORE

ALTERNATIVES
CAVEATS
AUTHOR
SEE ALSO

Memoize::ExpireFile – test for Memoize expiration semantics

DESCRIPTION

Memoize::ExpireTest – test for Memoize expiration semantics

DESCRIPTION

Memoize::NDBM_File – glue to provide EXISTS for NDBM_File for

Storable use

DESCRIPTION

Memoize::SDBM_File – glue to provide EXISTS for SDBM_File for

Storable use

DESCRIPTION

Memoize::Storable – store Memoized data in Storable database

DESCRIPTION

NDBM_File – Tied access to ndbm files

SYNOPSIS
DESCRIPTION

O_RDONLY, O_WRONLY, O_RDWR

DIAGNOSTICS

ndbm store returned -1, errno 22, key "..." at ...

BUGS AND WARNINGS

NEXT – Provide a pseudo-class NEXT that allows method redispach

SYNOPSIS
DESCRIPTION
Enforcing redispach
Avoiding repetitions

AUTHOR
BUGS AND IRRITATIONS
COPYRIGHT

Net::Cmd – Network Command class (as used by FTP, SMTP etc)

SYNOPSIS
DESCRIPTION
USER METHODS

debug (VALUE), message (), code (), ok (), status (), datasend (DATA), dataend ()

CLASS METHODS

debug_print (DIR, TEXT), debug_text (TEXT), command (CMD [, ARGS, ...]), unsupported (),
response (), parse_response (TEXT), getline (), ungetline (TEXT), read_until_dot (), tied_fh ()

EXPORTS
AUTHOR
COPYRIGHT

Net::Config – Local configuration data for libnet

SYNOPSIS
DESCRIPTION
METHODS

requires_firewall HOST

NetConfig VALUES

nnTP_hosts, snpp_hosts, pop3_hosts, smtp_hosts, ph_hosts, daytime_hosts, time_hosts, inet_domain,
ftp_firewall, ftp_firewall_type, ftp_ext_passive, ftp_int_pasive, local_netmask, test_hosts, test_exists

Net::Domain – Attempt to evaluate the current host's internet name

and domain

SYNOPSIS
DESCRIPTION

hostfqdn (), hostname (), hostdomain ()

AUTHOR
COPYRIGHT

Net::FTP – FTP Client class

SYNOPSIS
DESCRIPTION
OVERVIEW
CONSTRUCTOR

new (HOST [,OPTIONS])

METHODS

login ([LOGIN [,PASSWORD [, ACCOUNT]]]), authorize ([AUTH [, RESP]]), site (ARGS), type (TYPE [, ARGS]), ascii ([ARGS]) binary([ARGS]) ebcDic([ARGS]) byte([ARGS]), rename (OLDNAME, NEWNAME), delete (FILENAME), cwd ([DIR]), cdup (), pwd (), restart (WHERE), rmdir (DIR), mkdir (DIR [, RECURSE]), ls ([DIR]), dir ([DIR]), get (REMOTE_FILE [, LOCAL_FILE [, WHERE]]), put (LOCAL_FILE [, REMOTE_FILE]), put_unique (LOCAL_FILE [, REMOTE_FILE]), append (LOCAL_FILE [, REMOTE_FILE]), unique_name (), mdtm (FILE), size (FILE), supported (CMD), hash ([FILEHANDLE_GLOB_REF],[BYTES_PER_HASH_MARK]), nlst ([DIR]), list ([DIR]), retr (FILE), stor (FILE), stou (FILE), appe (FILE), port ([PORT]), pasv (), pasv_xfer (SRC_FILE, DEST_SERVER [, DEST_FILE]), pasv_xfer_unique (SRC_FILE, DEST_SERVER [, DEST_FILE])

), pasv_wait (NON_PASV_SERVER), abort (), quit ()

Methods for the adventurous

quot (CMD [,ARGS])

THE dataconn CLASS

read (BUFFER, SIZE [, TIMEOUT]), write (BUFFER, SIZE [, TIMEOUT]), bytes_read (), abort (), close ()

UNIMPLEMENTED

ALLO, SMNT, HELP, MODE, SYST, STAT, STRU, REIN

REPORTING BUGS

AUTHOR

SEE ALSO

USE EXAMPLES

<http://www.csh.rit.edu/~adam/Progs/autoftp-2.0.tar.gz>

CREDITS

COPYRIGHT

Net::NNTP – NNTP Client class

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

new ([HOST] [, OPTIONS])

METHODS

article ([MSGID|MSGNUM], [FH]), body ([MSGID|MSGNUM], [FH]), head ([MSGID|MSGNUM], [FH]), articlefh ([MSGID|MSGNUM]), bodyfh ([MSGID|MSGNUM]), headfh ([MSGID|MSGNUM]), nntpstat ([MSGID|MSGNUM]), group ([GROUP]), ihave (MSGID [, MESSAGE]), last (), date (), postok (), authinfo (USER, PASS), list (), newgroups (SINCE [, DISTRIBUTIONS]), newnews (SINCE [, GROUPS [, DISTRIBUTIONS]]), next (), post ([MESSAGE]), postfh (), slave (), quit ()

Extension methods

newsgroups ([PATTERN]), distributions (), subscriptions (), overview_fmt (), active_times (), active ([PATTERN]), xgtitle (PATTERN), xhdr (HEADER, MESSAGE-SPEC), xover (MESSAGE-SPEC), xpath (MESSAGE-ID), xpat (HEADER, PATTERN, MESSAGE-SPEC), xrover, listgroup ([GROUP]), reader

UNSUPPORTED

DEFINITIONS

MESSAGE-SPEC, PATTERN, Examples, [^] -, *bdc, [0-9a-zA-Z], a??d

SEE ALSO

AUTHOR

COPYRIGHT

Net::POP3 – Post Office Protocol 3 Client class (RFC1939)

SYNOPSIS

DESCRIPTION

EXAMPLES

CONSTRUCTOR

new ([HOST,] [OPTIONS])

METHODS

```
user ( USER ), pass ( PASS ), login ( [ USER [, PASS ] ] ), atop ( [ USER [, PASS ] ] ), top (
MSGNUM [, NUMLINES ] ), list ( [ MSGNUM ] ), get ( MSGNUM [, FH ] ), getfh ( MSGNUM ),
last ( ), popstat ( ), ping ( USER ), uidl ( [ MSGNUM ] ), delete ( MSGNUM ), reset ( ), quit ( )
```

NOTES

SEE ALSO

AUTHOR

COPYRIGHT

Net::Ping – check a remote host for reachability

SYNOPSIS

DESCRIPTION

Functions

```
Net::Ping->new([ $proto [, $def_timeout [, $bytes] ] ] );, $p->ping($host [,
$timeout] );, $p->source_verify( { 0 | 1 } );, $p->hires( { 0 | 1 } );, $p->bind($local_addr);,
$p->open($host);, $p->close();, $p->pingecho($host [, $timeout] );
```

WARNING

NOTES

INSTALL

AUTHORS

COPYRIGHT

Net::SMTP – Simple Mail Transfer Protocol Client

SYNOPSIS

DESCRIPTION

EXAMPLES

CONSTRUCTOR

```
new Net::SMTP [ HOST, ] [ OPTIONS ]
```

METHODS

```
banner ( ), domain ( ), hello ( DOMAIN ), etrn ( DOMAIN ), auth ( USERNAME, PASSWORD ),
mail ( ADDRESS [, OPTIONS] ), send ( ADDRESS ), send_or_mail ( ADDRESS ), send_and_mail (
ADDRESS ), reset ( ), recipient ( ADDRESS [, ADDRESS [ ... ] [, OPTIONS ] ] ), to ( ADDRESS [,
ADDRESS [ ... ] ], cc ( ADDRESS [, ADDRESS [ ... ] ], bcc ( ADDRESS [, ADDRESS [ ... ] ], data ( [
DATA ] ), expand ( ADDRESS ), verify ( ADDRESS ), help ( [ $subject ] ), quit ( )
```

ADDRESSES

SEE ALSO

AUTHOR

COPYRIGHT

Net::Time – time and daytime network client interface

SYNOPSIS

DESCRIPTION

```
inet_time ( [HOST [, PROTOCOL [, TIMEOUT]]]), inet_daytime ( [HOST [, PROTOCOL [,
TIMEOUT]]])
```

AUTHOR

COPYRIGHT

Net::hostent – by-name interface to Perl's built-in gethost*()

functions

SYNOPSIS
DESCRIPTION
EXAMPLES
NOTE
AUTHOR

Net::libnetFAQ, libnetFAQ – libnet Frequently Asked Questions

DESCRIPTION

Where to get this document
How to contribute to this document
Author and Copyright Information
Disclaimer
Obtaining and installing libnet
What is libnet ?
Which version of perl do I need ?
What other modules do I need ?
What machines support libnet ?
Where can I get the latest libnet release

Using Net::FTP

How do I download files from an FTP server ?
How do I transfer files in binary mode ?
How can I get the size of a file on a remote FTP server ?
How can I get the modification time of a file on a remote FTP server ?

How can I change the permissions of a file on a remote server ?
Can I do a reget operation like the ftp command ?
How do I get a directory listing from an FTP server ?
Changing directory to "" does not fail ?
I am behind a SOCKS firewall, but the Firewall option does not work ?
I am behind an FTP proxy firewall, but cannot access machines outside ?

My ftp proxy firewall does not listen on port 21
Is it possible to change the file permissions of a file on an FTP server ?

I have seen scripts call a method message, but cannot find it documented ?

Why does Net::FTP not implement mput and mget methods

Using Net::SMTP

Why can't the part of an Email address after the @ be used as the hostname ?

Why does Net::SMTP not do DNS MX lookups ?
The verify method always returns true ?
Debugging scripts
How can I debug my scripts that use Net::* modules ?
AUTHOR AND COPYRIGHT

Net::netent – by-name interface to Perl's built-in getnet* ()

functions

SYNOPSIS
DESCRIPTION
EXAMPLES
NOTE
AUTHOR

Net::protoent – by-name interface to Perl's built-in getproto* ()

functions

SYNOPSIS
DESCRIPTION
NOTE
AUTHOR

Net::servent – by-name interface to Perl's built-in getserv* ()

functions

SYNOPSIS
DESCRIPTION
EXAMPLES
NOTE
AUTHOR

Netrc, Net::Netrc – OO interface to users netrc file

SYNOPSIS
DESCRIPTION
THE .netrc FILE

machine name, default, login name, password string, account string, macdef name

CONSTRUCTOR

lookup (MACHINE [, LOGIN])

METHODS

login (), password (), account (), lpa ()

AUTHOR
SEE ALSO
COPYRIGHT

O – Generic interface to Perl Compiler backends

SYNOPSIS
DESCRIPTION
CONVENTIONS
IMPLEMENTATION
BUGS
AUTHOR

ODBM_File – Tied access to odbm files

SYNOPSIS
DESCRIPTION

O_RDONLY, O_WRONLY, O_RDWR

DIAGNOSTICS

odbm store returned -1, errno 22, key "..." at ...

BUGS AND WARNINGS

Opcode – Disable named opcodes when compiling perl code

SYNOPSIS

DESCRIPTION

NOTE

WARNING

Operator Names and Operator Lists

an operator name (opname), an operator tag name (optag), a negated opname or optag, an operator set (opset)

Opcode Functions

opcodes, opset (OP, ...), opset_to_ops (OPSET), opset_to_hex (OPSET), full_opset, empty_opset, invert_opset (OPSET), verify_opset (OPSET, ...), define_optag (OPTAG, OPSET), opmask_add (OPSET), opmask, opdesc (OP, ...), opdump (PAT)

Manipulating Opsets

TO DO (maybe)

Predefined Opcode Tags

:base_core, :base_mem, :base_loop, :base_io, :base_orig, :base_math, :base_thread, :default, :filesystem_read, :sys_db, :browse, :filesystem_open, :filesystem_write, :subprocess, :ownprocess, :others, :still_to_be_decided, :dangerous

SEE ALSO

AUTHORS

Opcode::Safe, Safe – Compile and execute code in restricted

compartments

SYNOPSIS

DESCRIPTION

a new namespace, an operator mask

WARNING

RECENT CHANGES

Methods in class Safe

permit (OP, ...), permit_only (OP, ...), deny (OP, ...), deny_only (OP, ...), trap (OP, ...), untrap (OP, ...), share (NAME, ...), share_from (PACKAGE, ARRAYREF), varglob (VARNAME), reval (STRING), rdo (FILENAME), root (NAMESPACE), mask (MASK)

Some Safety Issues

Memory, CPU, Snooping, Signals, State Changes

AUTHOR

Opcode::ops, ops – Perl pragma to restrict unsafe operations when

compiling

SYNOPSIS

DESCRIPTION

SEE ALSO

POSIX – Perl interface to IEEE Std 1003.1

SYNOPSIS

DESCRIPTION

NOTE

CAVEATS FUNCTIONS

_exit, abort, abs, access, acos, alarm, asctime, asin, assert, atan, atan2, atexit, atof, atoi, atol, bsearch, calloc, ceil, chdir, chmod, chown, clearerr, clock, close, closedir, cos, cosh, creat, ctermid, ctime, cuserid, difftime, div, dup, dup2, errno, execl, execle, execlp, execv, execve, execvp, exit, exp, fabs, fclose, fcntl, fdopen, feof, ferror, fflush, fgetc, fgetpos, fgets, fileno, floor, fmod, fopen, fork, fpathconf, fprintf, fputc, fputs, fread, free, freopen, frexp, fscanf, fseek, fsetpos, fstat, ftell, fwrite,getc, getchar, getcwd, getegid, getenv, geteuid, getgid, getgrgid, getgrnam, getgroups, getlogin, getpgrp, getpid, getppid, getpwnam, getpwuid, gets, getuid, gmtime, isalnum, isalpha, isatty, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, kill, labs, ldexp, ldiv, link, localeconv, localtime, log, log10, longjmp, lseek, malloc, mblen, mbstowcs, mbtowc, memchr, memcmp, memcpy, memmove, memset, mkdir, mkfifo, mktime, modf, nice, offsetof, open, opendir, pathconf, pause, perror, pipe, pow, printf, putc, putchar, puts, qsort, raise, rand, read, readdir, realloc, remove, rename, rewind, rewinddir, rmdir, scanf, setgid, setjmp, setlocale, setpgid, setsid, setuid, sigaction, siglongjmp, sigpending, sigprocmask, sigsetjmp, sigsuspend, sin, sinh, sleep, sprintf, sqrt, srand, sscanf, stat, strcat, strchr, strcmp, strcoll, strcpy, strcspn, strerror, strftime, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtod, strtok, strtol, strtoul, strxfrm, sysconf, system, tan, tanh, tcdrain, tcflow, tcflush, tcgetpgrp, tcsendbreak, tcsetpgrp, time, times, tmpfile, tmpnam, tolower, toupper, ttyname, tzname, tzset, umask, uname, ungetc, unlink, utime, vfprintf, vprintf, vsprintf, wait, waitpid, wcstombs, wctomb, write

CLASSES

POSIX::SigAction

new

POSIX::SigSet

new, addset, delset, emptyset, fillset, ismember

POSIX::Termios

new, getattr, getcc, getcflag, getiflag, getispeed, getlflag, getoflag, getospeed, setattr, setcc, setcflag, setiflag, setispeed, setlflag, setoflag, setospeed, Baud rate values, Terminal interface values, c_cc field values, c_cflag field values, c_iflag field values, c_lflag field values, c_oflag field values

PATHNAME CONSTANTS

Constants

POSIX CONSTANTS

Constants

SYSTEM CONFIGURATION

Constants

ERRNO

Constants

FCNTL

Constants

FLOAT

Constants

LIMITS

Constants

LOCALE

Constants

MATH

Constants

SIGNAL

Constants

STAT

Constants, Macros

STDLIB

Constants

STDIO

Constants

TIME

Constants

UNISTD

Constants

WAIT

Constants, WNOHANG, WUNTRACED, Macros, WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG

PerlIO – On demand loader for PerlIO layers and root of PerlIO::*

name space

SYNOPSIS**DESCRIPTION**

unix, stdio, perlio, crlf, utf8, bytes, raw, pop

Alternatives to raw

Defaults and how to override them

AUTHOR**SEE ALSO****PerlIO::encoding – encoding layer****SYNOPSIS****DESCRIPTION****SEE ALSO****PerlIO::scalar – support module for in-memory IO.****SYNOPSIS****DESCRIPTION****PerlIO::via – Helper class for PerlIO layers implemented in perl****SYNOPSIS****DESCRIPTION****EXPECTED METHODS**

```

$class-PUSHED([$mode[$fh]]), $obj-POPPED([$fh]), $obj-OPEN($path,$mode[$fh]),
$obj-BINMODE([$fh]), $obj-FDOPEN($fd,$fh),
$obj-SYSOPEN($path,$mode,$perm,[$fh]), $obj-FILENO($fh),
$obj-READ($buffer,$len,$fh), $obj-WRITE($buffer,$fh), $obj-FILL($fh), $obj-CLOSE($fh),
$obj-SEEK($posn,$whence,$fh), $obj-TELL($fh), $obj-UNREAD($buffer,$fh),
$obj-FLUSH($fh), $obj-SETLINEBUF($fh), $obj-CLEARERR($fh), $obj-ERROR($fh),
$obj-EOF($fh)

```

EXAMPLES

Example – a Hexadecimal Handle

PerlIO::via::QuotedPrint – PerlIO layer for quoted–printable strings

SYNOPSIS

DESCRIPTION

SEE ALSO

COPYRIGHT

Pod::Checker, podchecker () – check pod documents for syntax errors

SYNOPSIS

OPTIONS/ARGUMENTS

```
podchecker ()
    -warnings => val
```

DESCRIPTION

DIAGNOSTICS

Errors

empty =headn, =over on line *N* without closing =back, =item without previous =over, =back without previous =over, No argument for =begin, =end without =begin, Nested =begin's, =for without formatter specification, unresolved internal link *NAME*, Unknown command "*CMD*", Unknown interior–sequence "*SEQ*", nested commands *CMD*<...*CMD*<...>...>, garbled entity *STRING*, Entity number out of range, malformed link *L*<>, nonempty *Z*<>, empty *X*<>, Spurious text after =pod / =cut, Spurious character(s) after =back

Warnings

multiple occurrence of link target *name*, line containing nothing but whitespace in paragraph, file does not start with =head, previous =item has no contents, preceding non–item paragraph(s), =item type mismatch (*one* vs. *two*), *N* unescaped <> in paragraph, Unknown entity, No items in =over, No argument for =item, empty section in previous paragraph, Verbatim paragraph in *NAME* section

Hyperlinks

ignoring leading/trailing whitespace in link, (section) in '\$page' deprecated, alternative text/node '%s' contains non–escaped | or /

RETURN VALUE

EXAMPLES

INTERFACE

```
Pod::Checker->new( %options )

$checker->poderror( @args ), $checker->poderror( {%opts}, @args )

$checker->num_errors()

$checker->name()

$checker->node()

$checker->idx()

$checker->hyperlink()
```

AUTHOR

Pod::Find – find POD documents in directory trees

SYNOPSIS

DESCRIPTION

```
pod_find( { %opts } , @directories )
    -verbose => 1, -perl => 1, -script => 1, -inc => 1

simplify_name( $str )
pod_where( { %opts }, $pod )
    -inc => 1, -dirs => [ $dir1, $dir2, ... ], -verbose => 1

contains_pod( $file , $verbose )
AUTHOR
SEE ALSO
```

Pod::Html – module to convert pod files to HTML

SYNOPSIS

DESCRIPTION

ARGUMENTS

backlink, cachedir, css, flush, header, help, htmldir, htmlroot, index, infile, libpods, netscape, outfile, podpath, podroot, quiet, recurse, title, verbose

EXAMPLE

ENVIRONMENT

AUTHOR

SEE ALSO

COPYRIGHT

Pod::InputObjects – objects representing POD input paragraphs,

commands, etc.

SYNOPSIS

REQUIRES

EXPORTS

DESCRIPTION

package **Pod::InputSource**, package **Pod::Paragraph**, package **Pod::InteriorSequence**, package **Pod::ParseTree**

Pod::InputSource

new()

name()

handle()

was_cutting()

Pod::Paragraph

Pod::Paragraph->new()

\$pod_para->cmd_name()

\$pod_para->text()

\$pod_para->raw_text()

\$pod_para->cmd_prefix()

\$pod_para->cmd_separator()

\$pod_para->parse_tree()

\$pod_para->file_line()

Pod::InteriorSequence

Pod::InteriorSequence->new()

\$pod_seq->cmd_name()

\$pod_seq->prepend()

\$pod_seq->append()

```
$pod_seq->nested()  
$pod_seq->raw_text()  
$pod_seq->left_delimiter()  
$pod_seq->right_delimiter()  
$pod_seq->parse_tree()  
$pod_seq->file_line()  
Pod::InteriorSequence::DESTROY()  
Pod::ParseTree  
Pod::ParseTree->new()  
$ptree->top()  
$ptree->children()  
$ptree->prepend()  
$ptree->append()  
$ptree->raw_text()  
Pod::ParseTree::DESTROY()  
SEE ALSO  
AUTHOR
```

Pod::LaTeX – Convert Pod data to formatted Latex

```
SYNOPSIS  
DESCRIPTION  
OBJECT METHODS  
    initialize  
  
Data Accessors  
    AddPreamble  
  
AddPostamble  
  
Head1Level  
  
Label  
  
LevelNoNum  
  
MakeIndex  
  
ReplaceNAMEwithSection  
  
StartWithNewPage  
  
TableOfContents  
  
UniqueLabels  
  
UserPreamble  
  
UserPostamble  
  
Lists  
  
Subclassed methods  
  
begin_pod  
end_pod  
command  
verbatim  
textblock
```


interior_sequence

List Methods

begin_list

end_list

add_item

Methods for headings

head

Internal methods

_output

_replace_special_chars

_replace_special_chars_late

_create_label

_create_index

_clean_latex_commands

_split_delimited

NOTES

SEE ALSO

AUTHORS

COPYRIGHT

REVISION

Pod::Man – Convert POD data to formatted *roff input

SYNOPSIS

DESCRIPTION

center, date, fixed, fixedbold, fixeditalic, fixedbolditalic, name, quotes, release, section

DIAGNOSTICS

roff font should be 1 or 2 chars, not "%s", Invalid link %s, Invalid quote specification "%s", %s:%d:
Unknown command paragraph "%s", %s:%d: Unknown escape E<%s>, %s:%d: Unknown formatting
code %s, %s:%d: Unmatched

BUGS

CAVEATS

SEE ALSO

AUTHOR

COPYRIGHT AND LICENSE

Pod::ParseLink – Parse an L<> formatting code in POD text

SYNOPSIS

DESCRIPTION

SEE ALSO

AUTHOR

COPYRIGHT AND LICENSE

Pod::ParseUtils – helpers for POD parsing and conversion

SYNOPSIS

DESCRIPTION

Pod::List

```
Pod::List->new()  
  
$list->file()  
$list->start()  
$list->indent()  
$list->type()  
$list->rx()  
$list->item()  
$list->parent()  
$list->tag()
```

Pod::Hyperlink

```
Pod::Hyperlink->new()  
  
$link->parse($string)  
$link->markup($string)  
$link->text()  
$link->warning()  
$link->file(), $link->line()  
$link->page()  
$link->node()  
$link->alttext()  
$link->type()  
$link->link()
```

Pod::Cache

```
Pod::Cache->new()  
  
$cache->item()  
$cache->find_page($name)
```

Pod::Cache::Item

```
Pod::Cache::Item->new()  
  
$cacheitem->page()  
$cacheitem->description()  
$cacheitem->path()  
$cacheitem->file()  
$cacheitem->nodes()  
$cacheitem->find_node($name)  
$cacheitem->idx()
```

AUTHOR
SEE ALSO

Pod::Parser – base class for creating POD filters and translators

SYNOPSIS

REQUIRES

EXPORTS

DESCRIPTION

QUICK OVERVIEW

PARSING OPTIONS

-want_nonPODs (default: unset), **-process_cut_cmd** (default: unset), **-warnings** (default: unset)

RECOMMENDED SUBROUTINE/METHOD OVERRIDES

command()

\$cmd, \$text, \$line_num, \$pod_para

verbatim()

\$text, \$line_num, \$pod_para

textblock()

\$text, \$line_num, \$pod_para

interior_sequence()

OPTIONAL SUBROUTINE/METHOD OVERRIDES

new()**initialize()****begin_pod()****begin_input()****end_input()****end_pod()****preprocess_line()****preprocess_paragraph()**

METHODS FOR PARSING AND PROCESSING

parse_text()**-expand_seq** => *code-ref|method-name*, **-expand_text** => *code-ref|method-name*, **-expand_ptree**
=> *code-ref|method-name***interpolate()****parse_paragraph()****parse_from_filehandle()****parse_from_file()**

ACCESSOR METHODS

errorsub()**cutting()****parseopts()****output_file()****output_handle()****input_file()****input_handle()****input_streams()****top_stream()**

PRIVATE METHODS AND DATA

_push_input_stream()**_pop_input_stream()**

TREE-BASED PARSING

SEE ALSO

AUTHOR

Pod::Plainer – Perl extension for converting Pod to old style Pod.

SYNOPSIS

DESCRIPTION

EXPORT

AUTHOR

SEE ALSO

Pod::Select, podselect() – extract selected sections of POD from

input

SYNOPSIS

REQUIRES

EXPORTS

DESCRIPTION

SECTION SPECIFICATIONS

RANGE SPECIFICATIONS

OBJECT METHODS

`curr_headings()`

`select()`

`add_selection()`

`clear_selections()`

`match_section()`

`is_selected()`

EXPORTED FUNCTIONS

`podselect()`

`-output, -sections, -ranges`

PRIVATE METHODS AND DATA

`_compile_section_spec()`

`$self->{_SECTION_HEADINGS}`

`$self->{_SELECTED_SECTIONS}`

SEE ALSO

AUTHOR

Pod::Text – Convert POD data to formatted ASCII text

SYNOPSIS

DESCRIPTION

`alt, code, indent, loose, quotes, sentence, width`

DIAGNOSTICS

Bizarre space in item, Item called without tag, Can't open %s for reading: %s, Invalid quote specification "%s", %s:%d: Unknown command paragraph: %s, %s:%d: Unknown escape: %s, %s:%d: Unknown formatting code: %s, %s:%d: Unmatched =back

RESTRICTIONS

NOTES

SEE ALSO

AUTHOR

COPYRIGHT AND LICENSE

Pod::Text::Color – Convert POD data to formatted color ASCII text

SYNOPSIS

DESCRIPTION

BUGS
 SEE ALSO
 AUTHOR
 COPYRIGHT AND LICENSE

Pod::Text::Overstrike – Convert POD data to formatted overstrike

text

SYNOPSIS
 DESCRIPTION
 BUGS
 SEE ALSO
 AUTHOR
 COPYRIGHT AND LICENSE

Pod::Text::Termcap – Convert POD data to ASCII text with format

escapes

SYNOPSIS
 DESCRIPTION
 NOTES
 SEE ALSO
 AUTHOR
 COPYRIGHT AND LICENSE

Pod::Usage, pod2usage() – print a usage message from embedded pod

documentation

SYNOPSIS
 ARGUMENTS

-message, -msg, -exitval, -verbose, -output, -input, -pathlist

DESCRIPTION
 EXAMPLES
 Recommended Use
 CAVEATS
 AUTHOR
 ACKNOWLEDGEMENTS

Pod::t::basic, basic.pod – Test of various basic POD features in

translators.

HEADINGS

This *is* a "level 1" heading

 "Level" "2 heading

This *is* a "level 1" heading

 "Level" 2 heading

LINKS

OVER AND ITEMS

 This is a test, a, b, a, b, c, d, "foo", **bar**, baz, Some longer item text

FORMATTING CODES

 &, , <, >, ",

VERBATIM

CONCLUSION

Pod::t::htmlesc, Escape Sequences Test

DESCRIPTION

Pod::t::htmlview, Test HTML Rendering

SYNOPSIS

DESCRIPTION

METHODS => OTHER STUFF

new()

foo, bar, baz, Black Cat, Sat *on* the, Mat<!E<gt, 1 Cat, 2 Sat, 3 Mat

old()

TESTING FOR AND BEGIN

TESTING URLs hyperlinking

SEE ALSO

SDBM_File – Tied access to sdbm files

SYNOPSIS

DESCRIPTION

O_RDONLY, O_WRONLY, O_RDWR

DIAGNOSTICS

sdbm store returned -1, errno 22, key "..." at ...

BUGS AND WARNINGS

Safe – Compile and execute code in restricted compartments

SYNOPSIS

DESCRIPTION

a new namespace, an operator mask

WARNING

RECENT CHANGES

Methods in class Safe

permit (OP, ...), permit_only (OP, ...), deny (OP, ...), deny_only (OP, ...), trap (OP, ...), untrap (OP, ...), share (NAME, ...), share_from (PACKAGE, ARRAYREF), varglob (VARNAME), reval (STRING), rdo (FILENAME), root (NAMESPACE), mask (MASK)

Some Safety Issues

Memory, CPU, Snooping, Signals, State Changes

AUTHOR

Scalar::Util – A selection of general-utility scalar subroutines

SYNOPSIS

DESCRIPTION

blessed EXPR, dualvar NUM, STRING, isweak EXPR, openhandle FH, reftype EXPR, tainted EXPR, weaken REF

KNOWN BUGS

COPYRIGHT

BLATANT PLUG

Search::Dict, look – search for key in dictionary file

SYNOPSIS

DESCRIPTION

SelectSaver – save and restore selected file handle

SYNOPSIS

DESCRIPTION

SelfLoader – load functions only on demand

SYNOPSIS

DESCRIPTION

The `__DATA__` token

SelfLoader autoloading

Autoloading and package lexicals

SelfLoader and AutoLoader

`__DATA__`, `__END__`, and the `FOOBAR::DATA` filehandle.

Classes and inherited methods.

Multiple packages and fully qualified subroutine names

Shell – run shell commands transparently within perl

SYNOPSIS

DESCRIPTION

OBJECT ORIENTED SYNTAX

AUTHOR

Socket, sockaddr_in, sockaddr_un, inet_aton, inet_ntoa – load the C

socket.h defines and structure manipulators

SYNOPSIS

DESCRIPTION

`inet_aton` HOSTNAME, `inet_ntoa` IP_ADDRESS, `INADDR_ANY`, `INADDR_BROADCAST`, `INADDR_LOOPBACK`, `INADDR_NONE`, `sockaddr_family` SOCKADDR, `sockaddr_in` PORT, ADDRESS, `sockaddr_in` SOCKADDR_IN, `pack_sockaddr_in` PORT, IP_ADDRESS, `unpack_sockaddr_in` SOCKADDR_IN, `sockaddr_un` PATHNAME, `sockaddr_un` SOCKADDR_UN, `pack_sockaddr_un` PATH, `unpack_sockaddr_un` SOCKADDR_UN

Storable – persistence for Perl data structures

SYNOPSIS

DESCRIPTION

MEMORY STORE

ADVISORY LOCKING

SPEED

CANONICAL REPRESENTATION

FORWARD COMPATIBILITY

utf8 data, restricted hashes, files from future versions of Storable

ERROR REPORTING

WIZARDS ONLY

Hooks

`STORABLE_freeze obj, cloning`, `STORABLE_thaw obj, cloning, serialized, ..`

Predicates

`Storable::last_op_in_netorder`, `Storable::is_storing`,
`Storable::is_retrieving`

Recursion

Deep Cloning

Storable magic
EXAMPLES
WARNING
BUGS
 64 bit data in perl 5.6.0 and 5.6.1
CREDITS
AUTHOR
SEE ALSO

Switch – A switch statement for Perl

VERSION
SYNOPSIS
BACKGROUND
DESCRIPTION
 Allowing fall-through
 Automating fall-through
 Alternative syntax
 Higher-order Operations
DEPENDENCIES
AUTHOR
BUGS
LIMITATION
COPYRIGHT

Symbol – manipulate Perl symbols and their names

SYNOPSIS
DESCRIPTION

Sys::Hostname – Try every conceivable way to get hostname

SYNOPSIS
DESCRIPTION
AUTHOR

Syslog, Sys::Syslog, openlog, closelog, setlogmask, syslog – Perl

interface to the UNIX syslog(3) calls

SYNOPSIS
DESCRIPTION

 openlog \$ident, \$logopt, \$facility, syslog \$priority, \$format, @args, setlogmask
 \$mask_priority, setlogsock \$sock_type [\$stream_location] (added in 5.004_02),
 closelog

EXAMPLES
SEE ALSO
AUTHOR

Syslog::Syslog, Sys::Syslog, openlog, closelog, setlogmask, syslog –

Perl interface to the UNIX syslog(3) calls

SYNOPSIS
DESCRIPTION

 openlog \$ident, \$logopt, \$facility, syslog \$priority, \$format, @args, setlogmask
 \$mask_priority, setlogsock \$sock_type [\$stream_location] (added in 5.004_02),
 closelog

EXAMPLES
SEE ALSO
AUTHOR

Term::ANSIColor – Color screen output using ANSI escape sequences

SYNOPSIS
DESCRIPTION
DIAGNOSTICS

Bad escape sequence %s, Bareword "%s" not allowed while "strict subs" in use, Invalid attribute name %s, Name "%s" used only once: possible typo, No comma allowed after filehandle, No name for escape sequence %s

ENVIRONMENT
ANSI_COLORS_DISABLED

RESTRICTIONS
NOTES
SEE ALSO
AUTHORS
COPYRIGHT AND LICENSE

Term::Cap – Perl termcap interface

SYNOPSIS
DESCRIPTION
METHODS

Tgetent, OSPEED, TERM

Tpad, \$string, \$cnt, \$FH

Tputs, \$cap, \$cnt, \$FH

Tgoto, \$cap, \$col, \$row, \$FH

Trequire

EXAMPLES
COPYRIGHT AND LICENSE
AUTHOR
SEE ALSO

Term::Complete – Perl word completion module

SYNOPSIS
DESCRIPTION

<tab>, ^D, ^U, , <bs>

DIAGNOSTICS
BUGS
AUTHOR

Term::ReadLine – Perl interface to various readline packages. If

no real package is found, substitutes stubs instead of basic functions.

SYNOPSIS
DESCRIPTION

Minimal set of supported functions

ReadLine, new, readline, addhistory, IN, \$OUT, MinLine, findConsole, Attribs, Features

Additional supported functions

tkRunning, ornaments, newTTY

EXPORTS

ENVIRONMENT

CAVEATS

Test – provides a simple framework for writing test scripts

SYNOPSIS

DESCRIPTION

Functions

plan

_to_value

ok

TEST TYPES

NORMAL TESTS, SKIPPED TESTS, TODO TESTS

ONFAIL

BUGS and CAVEATS

NOTE

SEE ALSO

AUTHOR

Test::Builder – Backend for building test libraries

SYNOPSIS

DESCRIPTION

Construction

new

Setting up tests

exported_to

plan

expected_tests

no_plan

skip_all

Running tests

ok

is_eq, is_num

isnt_eq, isnt_num

like, unlike

maybe_regex

cmp_ok

BAILOUT

skip

todo_skip

skip_rest

Test style

level**use_numbers****no_header, no_ending**

Output

diag**_print****output, failure_output, todo_output**

Test Status and Info

current_test**summary****details** *UNIMPLEMENTED*, **todo****caller****_sanity_check****_whoa****_my_exit**

THREADS

EXAMPLES

SEE ALSO

AUTHORS

COPYRIGHT

Test::Harness – run perl standard test scripts with statistics

SYNOPSIS

DESCRIPTION

The test script output

‘1..M’, ‘ok’, ‘not ok’. Ok?, test numbers, test names, Skipping tests, Todo tests, Bail out!, Comments, Anything else

Taint mode

Configuration variables.

\$Test::Harness::verbose, \$Test::Harness::switches

Failure

Failed Test, Stat, Wstat, Total, Fail, Failed, List of Failed

Functions

runtests**_all_ok****_globdir****_run_all_tests****_mk_leader****_leader_width**

EXPORT

DIAGNOSTICS

```
All tests successful.\nFiles=%d, Tests=%d, %s,FAILED tests
%s\n\tFailed %d/%d tests, %.2f%% okay.,Test returned status %d (wstat
%d),Failed 1 test, %.2f%% okay. %s,Failed %d/%d tests, %.2f%% okay.
%s,FAILED--Further testing stopped: %s
```

ENVIRONMENT

```
HARNESS_ACTIVE, HARNESS_COLUMNS, HARNESS_COMPILE_TEST,
HARNESS_FILELEAK_IN_DIR, HARNESS_IGNORE_EXITCODE, HARNESS_NOTTY,
HARNESS_PERL_SWITCHES, HARNESS_VERBOSE
```

EXAMPLE

SEE ALSO

AUTHORS

TODO

BUGS

Test::Harness::Assert – simple assert

SYNOPSIS

DESCRIPTION

Functions

assert

AUTHOR

SEE ALSO

Test::Harness::Iterator – Internal Test::Harness Iterator

SYNOPSIS

DESCRIPTION

Test::Harness::Straps – detailed analysis of test results

SYNOPSIS

DESCRIPTION

Construction

new**_init**

Analysis

analyze**analyze_fh****analyze_file****_switches****_INC2PERL5LIB****_filtered_INC****_restore_PERL5LIB**

Parsing

_is_comment**_is_header**

`_is_test`
`_is_bail_out`
`_reset_file_state`
Results
 `_detailize`
EXAMPLES
AUTHOR
SEE ALSO

Test::More – yet another framework for writing test scripts

SYNOPSIS
DESCRIPTION
 I love it when a plan comes together
Test names
I'm ok, you're not ok.
 ok
is, isnt
like
unlike
cmp_ok
can_ok
isa_ok
pass, fail
Diagnostics
 diag
Module tests
 use_ok
require_ok
Conditional tests
 SKIP: BLOCK
TODO: BLOCK, todo_skip
When do I use SKIP vs. TODO?
Comparison functions
 is_deeply
eq_array
eq_hash
eq_set
Extending and Embedding Test::More
 builder

NOTES

BUGS and CAVEATS

Making your own `ok()`, The `eq_*` family has some caveats, `Test::Harness` upgrades

HISTORY

SEE ALSO

AUTHORS

COPYRIGHT

Test::Simple – Basic utilities for writing tests.

SYNOPSIS

DESCRIPTION

`ok`

EXAMPLE

CAVEATS

NOTES

HISTORY

SEE ALSO

Test::More, Test, Test::Unit, Test::Inline, SelfTest, Test::Harness

AUTHORS

COPYRIGHT

Test::Tutorial – A tutorial about writing really basic tests

DESCRIPTION

Nuts and bolts of testing.

Where to start?

Names

Test the manual

Sometimes the tests are wrong

Testing lots of values

Informative names

Skipping tests

Todo tests

Testing with taint mode.

FOOTNOTES

AUTHORS

COPYRIGHT

Text::Abbrev, abbrev – create an abbreviation table from a list

SYNOPSIS

DESCRIPTION

EXAMPLE

Text::Balanced – Extract delimited text sequences from strings.

SYNOPSIS

DESCRIPTION

General behaviour in list contexts

`[0]`, `[1]`, `[2]`

General behaviour in scalar and void contexts

A note about prefixes

`extract_delimited`

extract_bracketed**extract_tagged**

```
reject => $listref, ignore => $listref, fail => $str, [0], [1], [2], [3],
[4], [5]
```

gen_extract_tagged**extract_quotelike**

```
[0], [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]
```

extract_quotelike and "here documents"

```
[0], [1], [2], [3], [4], [5], [6], [7..10]
```

extract_codeblock**extract_multiple****gen_delimited_pat****DIAGNOSTICS**

```
C<Did not find a suitable bracket: "%s">, C<Did not find prefix: /%s/>,
Did not find opening bracket after prefix: "%s", No quotelike
operator found after prefix: "%s", Unmatched closing bracket: "%c",
Unmatched opening bracket(s): "%s", Unmatched embedded quote (%s), Did
not find closing delimiter to match '%s', Mismatched closing bracket:
expected "%c" but found "%s", No block delimiter found after
quotelike "%s", Did not find leading dereferencer, Bad identifier after
dereferencer, Did not find expected opening bracket at %s, Improperly
nested codeblock at %s, Missing second block for quotelike "%s", No
match found for opening bracket, Did not find opening tag: /%s/, Unable
to construct closing tag to match: /%s/, Found invalid nested tag: %s,
Found unbalanced nested tag: %s, Did not find closing tag
```

AUTHOR**BUGS AND IRRITATIONS****COPYRIGHT****Text::ParseWords – parse text into an array of tokens or array of**

```
arrays
```

SYNOPSIS**DESCRIPTION****EXAMPLES****AUTHORS****Text::Soundex – Implementation of the Soundex Algorithm as Described**

```
by Knuth
```

SYNOPSIS**DESCRIPTION****EXAMPLES****LIMITATIONS****AUTHOR****Text::Tabs — expand and unexpand tabs per the unix expand(1) and**

```
unexpand(1)
```

SYNOPSIS**DESCRIPTION****BUGS**

AUTHOR

Text::Wrap – line wrapping to form simple paragraphs

SYNOPSIS
DESCRIPTION
OVERRIDES
EXAMPLE
AUTHOR

Thread – manipulate threads in Perl (for old code only)

CAVEAT
SYNOPSIS
DESCRIPTION
FUNCTIONS

`$thread = Thread-new(\&start_sub) , $thread = Thread-new(\&start_sub, LIST), lock
VARIABLE, async BLOCK;, Thread-self, cond_wait VARIABLE, cond_signal VARIABLE,
cond_broadcast VARIABLE, yield`

METHODS

`join, eval, detach, equal, tid, flags, done`

LIMITATIONS
SEE ALSO

Thread::Queue – thread-safe queues

SYNOPSIS
DESCRIPTION
FUNCTIONS AND METHODS

`new, enqueue LIST, dequeue, dequeue_nb, pending`

SEE ALSO

Thread::Semaphore – thread-safe semaphores

SYNOPSIS
DESCRIPTION
FUNCTIONS AND METHODS

`new, new NUMBER, down, down NUMBER, up, up NUMBER`

Thread::Signal – Start a thread which runs signal handlers reliably

(for old code)

CAVEAT
SYNOPSIS
DESCRIPTION
BUGS

Thread::Specific – thread-specific keys

SYNOPSIS
DESCRIPTION

Tie::Array – base class for tied arrays

SYNOPSIS
DESCRIPTION

`TIEARRAY classname, LIST, STORE this, index, value, FETCH this, index, FETCHSIZE this,
STORESIZE this, count, EXTEND this, count, EXISTS this, key, DELETE this, key, CLEAR this,
DESTROY this, PUSH this, LIST, POP this, SHIFT this, UNSHIFT this, LIST, SPLICE this, offset,`

length, LIST
 CAVEATS
 AUTHOR

Tie::File – Access the lines of a disk file via a Perl array

SYNOPSIS
 DESCRIPTION
 recsep
 autochomp
 mode
 memory
 dw_size
 Option Format
 Public Methods
 flock
 autochomp
 defer, flush, discard, and autodefer
 Tying to an already-opened filehandle
 Deferred Writing
 Autodeferring
 CAVEATS
 SUBCLASSING
 WHAT ABOUT DB_File?
 AUTHOR
 LICENSE
 WARRANTY
 THANKS
 TODO

Tie::Handle, Tie::StdHandle – base class definitions for tied

handles
 SYNOPSIS
 DESCRIPTION
 TIEHANDLE classname, LIST, WRITE this, scalar, length, offset, PRINT this, LIST, PRINTF this, format, LIST, READ this, scalar, length, offset, READLINE this, GETC this, CLOSE this, OPEN this, filename, BINMODE this, EOF this, TELL this, SEEK this, offset, whence, DESTROY this
 MORE INFORMATION
 COMPATIBILITY

Tie::Hash, Tie::StdHash, Tie::ExtraHash – base class definitions for

tied hashes
 SYNOPSIS
 DESCRIPTION
 TIEHASH classname, LIST, STORE this, key, value, FETCH this, key, FIRSTKEY this, NEXTKEY this, lastkey, EXISTS this, key, DELETE this, key, CLEAR this
 Inheriting from **Tie::StdHash**
 Inheriting from **Tie::ExtraHash**
 UNTIE and DESTROY
 MORE INFORMATION

Tie::Memoize – add data to hash when needed

SYNOPSIS
DESCRIPTION
Inheriting from **Tie::Memoize**
EXAMPLE
BUGS
AUTHOR

Tie::RefHash – use references as hash keys

SYNOPSIS
DESCRIPTION
EXAMPLE
AUTHOR
VERSION
SEE ALSO

Tie::Scalar, Tie::StdScalar – base class definitions for tied

scalars

SYNOPSIS
DESCRIPTION

TIESCALAR classname, LIST, FETCH this, STORE this, value, DESTROY this

MORE INFORMATION

Tie::SubstrHash – Fixed-table-size, fixed-key-length hashing

SYNOPSIS
DESCRIPTION
CAVEATS

Time::HiRes – High resolution alarm, sleep, gettimeofday, interval

timers

SYNOPSIS
DESCRIPTION

gettimeofday (), usleep (\$useconds), ualarm (\$useconds [, \$interval_useconds]),
tv_interval, time (), sleep (\$floating_seconds), alarm (\$floating_seconds [,
\$interval_floating_seconds]), setitimer, getitimer (\$which)

EXAMPLES
C API
CAVEATS
AUTHORS
REVISION
COPYRIGHT

Time::Local – efficiently compute time from local and GMT time

SYNOPSIS
DESCRIPTION
IMPLEMENTATION
BUGS

Time::gmtime – by-name interface to Perl's built-in gmtime ()

function

SYNOPSIS
DESCRIPTION
NOTE
AUTHOR

Time::localtime – by-name interface to Perl's built-in localtime()

function

SYNOPSIS
DESCRIPTION
NOTE
AUTHOR

Time::tm – internal object used by Time::gmtime and Time::localtime

SYNOPSIS
DESCRIPTION
AUTHOR

UNIVERSAL – base class for ALL classes (blessed references)

SYNOPSIS
DESCRIPTION

`$obj-isa(TYPE), CLASS-isa(TYPE), isa(VAL, TYPE), $obj-can(METHOD), CLASS-can(METHOD), can(VAL, METHOD), VERSION ([REQUIRE])`

Unicode::Collate – Unicode Collation Algorithm

SYNOPSIS
DESCRIPTION

Constructor and Tailoring

`alternate, backwards, entry, ignoreName, ignoreChar, level, normalization, overrideCJK, overrideHangul, preprocess, rearrange, table, undefName, undefChar, katakana_before_hiragana, upper_before_lower`

Methods for Collation

```
@sorted = $Collator->sort(@not_sorted), $result =
$Collator->cmp($a, $b), $result = $Collator->eq($a, $b), $result =
$Collator->ne($a, $b), $result = $Collator->lt($a, $b), $result =
$Collator->le($a, $b), $result = $Collator->gt($a, $b), $result =
$Collator->ge($a, $b), $sortKey = $Collator->getSortKey($string),
$sortKeyForm = $Collator->viewSortKey($string), $position =
$Collator->index($string, $substring), ($position, $length) =
$Collator->index($string, $substring)
```

Other Methods

`UCA_Version, Base_Unicode_Version`

EXPORT
TODO
CAVEAT
BUGS

AUTHOR
SEE ALSO

<http://www.unicode.org/unicode/reports/tr10/>, <http://www.unicode.org/unicode/reports/tr10/allkeys.txt>,
<http://www.unicode.org/unicode/reports/tr15/>, <http://www.unicode.org/unicode/reports/tr18/>,
[Unicode::Normalize](#)

Unicode::Normalize – Unicode Normalization Forms**SYNOPSIS****DESCRIPTION****Normalization Forms**

```
$NFD_string = NFD($string), $NFC_string = NFC($string),
$NFKD_string = NFKD($string), $NFKC_string = NFKC($string),
$normalized_string = normalize($form_name, $string)
```

Decomposition and Composition

```
$decomposed_string = decompose($string), $decomposed_string =
decompose($string, $useCompatMapping), $reordered_string =
reorder($string), $composed_string = compose($string)
```

Quick Check

```
$result = checkNFD($string), $result = checkNFC($string), $result
= checkNFKD($string), $result = checkNFKC($string), $result =
check($form_name, $string)
```

Character Data

```
$canonical_decomposed = getCanon($codepoint),
$compatibility_decomposed = getCompat($codepoint),
$codepoint_composite = getComposite($codepoint_here,
$codepoint_next), $combining_class = getCombinClass($codepoint),
$is_exclusion = isExclusion($codepoint), $is_singleton =
isSingleton($codepoint), $is_non_startar_decomposition =
isNonStDecomp($codepoint), $may_be_composed_with_prev_char =
isComp2nd($codepoint)
```

EXPORT**AUTHOR****SEE ALSO**

<http://www.unicode.org/unicode/reports/tr15/>,

<http://www.unicode.org/Public/UNIDATA/DerivedNormalizationProps.txt>

Unicode::UCD – Unicode character database**SYNOPSIS****DESCRIPTION**

charinfo

charblock

charscript

charblocks

charscripts

Blocks versus Scripts

Matching Scripts and Blocks

Code Point Arguments

charinrange

compexcl

casefold

casespec

Unicode::UCD::UnicodeVersion

Implementation Note

BUGS

AUTHOR

User::grent – by-name interface to Perl's built-in getgr* ()

functions

SYNOPSIS

DESCRIPTION

NOTE

AUTHOR

User::pwent – by-name interface to Perl's built-in getpw* ()

functions

SYNOPSIS

DESCRIPTION

System Specifics

NOTE

AUTHOR

HISTORY

March 18th, 2000

Win32 – Interfaces to some Win32 API Functions

DESCRIPTION

Alphabetical Listing of Win32 Functions

```
Win32::AbortSystemShutdown(MACHINE), Win32::BuildNumber(),
Win32::CopyFile(FROM, TO, OVERWRITE), Win32::DomainName(),
Win32::ExpandEnvironmentStrings(STRING), Win32::FormatMessage(ERRORCODE),
Win32::FsType(), Win32::FreeLibrary(HANDLE), Win32::GetArchName(),
Win32::GetChipName(), Win32::GetCwd(), Win32::GetFullPathName(FILENAME),
Win32::GetLastError(), Win32::GetLongPathName(PATHNAME),
Win32::GetNextAvailDrive(), Win32::GetOSVersion(),
Win32::GetOSName(), Win32::GetShortPathName(PATHNAME),
Win32::GetProcAddress(INSTANCE, PROCNAME), Win32::GetTickCount(),
Win32::InitiateSystemShutdown, Win32::IsWinNT(), Win32::IsWin95(),
Win32::LoadLibrary(LIBNAME), Win32::LoginName(),
Win32::LookupAccountName(SYSTEM, ACCOUNT, DOMAIN, SID, SIDTYPE),
Win32::LookupAccountSID(SYSTEM, SID, ACCOUNT, DOMAIN, SIDTYPE),
Win32::MsgBox(MESSAGE [, FLAGS [, TITLE]]), Win32::NodeName(),
Win32::RegisterServer(LIBRARYNAME), Win32::SetChildShowWindow(SHOWWINDOW),
Win32::SetCwd(NEWDIRECTORY), Win32::SetLastError(ERROR), Win32::Sleep(TIME),
Win32::Spawn(COMMAND, ARGS, PID), Win32::UnregisterServer(LIBRARYNAME)
```

XSLoader – Dynamically load C libraries into Perl code

SYNOPSIS

DESCRIPTION

AUTHOR

AUXILIARY DOCUMENTATION

Here should be listed all the extra programs' documentation, but they don't all have manual pages yet:

```
a2p
s2p
find2perl
h2ph
c2ph
h2xs
```

xsubpp
pod2man
wrapsuid

AUTHOR

Larry Wall <*larry@wall.org*>, with the help of oodles of other folks.

NAME

perltodo – Perl TO–DO List

DESCRIPTION

This is a list of wishes for Perl. Send updates to *perl5-porters@perl.org*. If you want to work on any of these projects, be sure to check the perl5-porters archives for past ideas, flames, and propaganda. This will save you time and also prevent you from implementing something that Larry has already vetoed. One set of archives may be found at:

<http://www.xray.mpe.mpg.de/mailling-lists/perl5-porters/>

To do during 5.6.x**Support for I/O disciplines**

perlio provides this, but the interface could be a lot more straightforward.

Autoload bytes.pm

When the lexer sees, for instance, `bytes : : length`, it should automatically load the `bytes` pragma.

Make "\u{XXXX}" et al work

Danger, Will Robinson! Discussing the semantics of `"\x{F00}"`, `"\xF00"` and `"\U{F00}"` on P5P *will* lead to a long and boring flamewar.

Create a char *sv_pvprintify(sv, STRLEN *lenp, UV flags)

For displaying PVs with control characters, embedded nulls, and Unicode. This would be useful for printing warnings, or data and regex dumping, `not_a_number()`, and so on.

Requirements: should handle both byte and UTF8 strings. `isPRINT()` characters printed as-is, character less than 256 as `\xHH`, Unicode characters as `\x{HHH}`. Don't assume ASCII-like, either, get somebody on EBCDIC to test the output.

Possible options, controlled by the flags: – whitespace (other than ' ' of `isPRINT()`) printed as-is – use `isPRINT_LC()` instead of `isPRINT()` – print control characters like this: `"\cA"` – print control characters like this: `"^A"` – non-PRINTables printed as `'.'` instead of `\xHH` – use `\OOO` instead of `\xHH` – use the C/Perl-metacharacters like `\n`, `\t` – have a maximum length for the produced string (read it from `*lenp`) – append a `"..."` to the produced string if the maximum length is exceeded – really fancy: print unicode characters as `\N{...}`

NOTE: `pv_display()`, `pv_uni_display()`, `sv_uni_display()` are already doing something like the above.

Overloadable regex assertions

This may or may not be possible with the current regular expression engine. The idea is that, for instance, `\b` needs to be algorithmically computed if you're dealing with Thai text. Hence, the `\b` assertion wants to be overloaded by a function.

Unicode

- Allow for long form of the General Category Properties, e.g `\p{IsOpenPunctuation}`, not just the abbreviated form, e.g. `\p{IsPs}`.
- Allow for the metaproperties: `XID_Start`, `XID_Continue`, `NF*_NO`, `NF*_MAYBE` (require the `DerivedCoreProperties` and `DerviceNormalizationProperties` files).

There are also multiple value properties still unimplemented: `Numeric Type`, `East Asian Width`.

- Case Mappings? <http://www.unicode.org/unicode/reports/tr21/>

Mostly implemented (all of 1:1, 1:N, N:1), only the "final sigma" and locale-specific rules of SpecCase are not implemented.

- UTF-8 identifier names should probably be canonicalized: NFC?
- UTF-8 in package names and sub names? The first is problematic because of the mapping to pathnames, ditto for the second one if one does autosplitting, for example. Some of this works already in 5.8.0, but essentially it is unsupported. Constructs to consider, at the very least:

```
use utf8;
package UnicodePackage;
sub new { bless {}, shift };
sub UnicodeMethod1 { ... $_[0]->UnicodeMethod2(...) ... }
sub UnicodeMethod2 { ... } # in here caller(0) should contain Unicode
...
package main;
my $x = UnicodePackage->new;
print ref $x, "\n"; # should be Unicode
$x->UnicodeMethod1(...);
my $y = UnicodeMethod3 UnicodePackage ...;
```

In the above all *UnicodeXxx* contain (identifier-worthy) characters beyond the code point 255, for example 256. Wherever package/class or subroutine names can be returned needs to be checked for Unicodeness.

See [UNICODE REGULAR EXPRESSION SUPPORT LEVEL](#) for what's there and what's missing. Almost all of Levels 2 and 3 is missing, and as of 5.8.0 not even all of Level 1 is there. They have some tricks Perl doesn't yet implement, such as character class subtraction.

<http://www.unicode.org/unicode/reports/tr18/>

Work out exit/die semantics for threads

There are some suggestions to use for example something like this: default to "(thread exiting first will) wait for the other threads until up to 60 seconds". Other possibilities:

```
use threads wait => 0;
```

Do not wait.

```
use threads wait_for => 10;
```

Wait up to 10 seconds.

```
use threads wait_for => -1;
```

Wait for ever.

<http://archive.develooper.com/perl5-porters@perl.org/msg79618.html>

Better support for nonpreemptive threading systems like GNU pth

To better support nonpreemptive threading systems, perhaps some of the blocking functions internally in Perl should do a `yield()` before a blocking call. (Now certain threads tests (`{basic,list,thread.t}`) simply do a `yield()` before they `sleep()` to give nonpreemptive thread implementations a chance).

In some cases, like the GNU pth, which has replacement functions that are nonblocking (`pth_select` instead of `select`), maybe Perl should be using them instead when built for threading.

Typed lexicals for compiler

Compiler workarounds for Win32

AUTOLOADing in the compiler

Fixing comppadlist when compiling

Cleaning up exported namespace

Complete signal handling

Add `PERL_ASYNC_CHECK` to opcodes which loop; replace `sigsetjmp` with `sigjmp`; check wait for signal safety.

Out-of-source builds

This was done for 5.6.0, but needs reworking for 5.7.x

POSIX realtime support

POSIX 1003.1 1996 Edition support—realtime stuff: POSIX semaphores, message queues, shared memory, realtime clocks, timers, signals (the metaconfig units mostly already exist for these)

UNIX98 support

Reader–writer locks, realtime/asynchronous IO

IPv6 Support

There are non–core modules, such as `Socket6`, but these will need integrating when IPv6 actually starts to really happen. See RFC 2292 and RFC 2553.

Long double conversion

Floating point formatting is still causing some weird test failures.

Locales

Locales and Unicode interact with each other in unpleasant ways. One possible solution would be to adopt/support ICU:

<http://oss.software.ibm.com/developerworks/opensource/icu/project/>

Arithmetic on non–Arabic numerals

[1234567890] aren't the only numerals any more.

POSIX Unicode character classes

([`=a=`] for equivalence classes, [`.ch.`] for collation.) These are dependent on Unicode normalization and collation.

Factoring out common suffices/prefices in regexps (trie optimization)

Currently, the user has to optimize `foo|far` and `foo|goo` into `f(?:oo|ar)` and `[fg]oo` by hand; this could be done automatically.

Security audit shipped utilities

All the code we ship with Perl needs to be sensible about temporary file handling, locking, input validation, and so on.

Sort out the uid–setting mess

Currently there are several problems with the setting of uids (`$<`, `$` for the real and effective uids). Firstly, what exactly `setuid()` call gets invoked in which platform is simply a big mess that needs to be untangled. Secondly, the effects are apparently not standard across platforms, (if you first set `$<` and then `$`, or vice versa, being uid == euid == zero, or just euid == zero, or as a normal user, what are the results?). The test suite not (usually) being run as root means that these things do not get much testing. Thirdly, there's quite often a third uid called saved uid, and Perl has no knowledge of that feature in any way. (If one has the saved uid of zero, one can get back any real and effective uids.) As an example, to change also the saved uid, one needs to set the real and effective uids **twice**— in most systems, that is: in HP–UX that doesn't seem to work.

Custom opcodes

Have a way to introduce user-defined opcodes without the subroutine call overhead of an XSUB; the user should be able to create PP code. Simon Cozens has some ideas on this.

DLL Versioning

Windows needs a way to know what version of an XS or libperl DLL it's loading.

Introduce @ (and @)

`$ (` may return "foo bar baz". Unfortunately, since groups can theoretically have spaces in their names, this could be one, two or three groups.

Floating point handling

NaN and inf support is particularly troublesome. (`fp_classify()`, `fp_class()`, `fp_class_d()`, `class()`, `isinf()`, `isfinite()`, `finite()`, `isnormal()`, `unordered()`, `<ieeefp.h`, `<fp_class.h` (there are metaconfig units for all these) (I think), `fp_setmask()`, `fp_getmask()`, `fp_setround()`, `fp_getround()` (no metaconfig units yet for these). Don't forget `finitel()`, `fp_classl()`, `fp_class_l()`, (yes, both do, unfortunately, exist), and `unorderedl()`.)

As of Perl 5.6.1, there is a Perl macro, `Perl_isnan()`.

IV/UV preservation

Nicholas Clark has done a lot of work on this, but work is continuing. `+`, `-` and `*` work, but guards need to be in place for `%`, `/`, `&`, `oct`, `hex` and `pack`.

Replace pod2html with something using Pod::Parser

The CPAN module `Marek::Pod::Html` may be a more suitable basis for a `pod2html` converter; the current one duplicates the functionality abstracted in `Pod::Parser`, which makes updating the POD language difficult.

Automate module testing on CPAN

When a new Perl is being beta tested, porters have to manually grab their favourite CPAN modules and test them – this should be done automatically.

sendmsg and recvmsg

We have all the other BSD socket functions but these. There are metaconfig units for these functions which can be added. To avoid these being new opcodes, a solution similar to the way `socketatmark` was added would be preferable. (Autoload the `IO::whatever` module.)

Rewrite perlre documentation

The new-style patterns need full documentation, and the whole document needs to be a lot clearer.

Convert example code to IO::Handle filehandles

Document Win32 choices

Check new modules

Make roffitall find pods and libs itself

Simon Cozens has done some work on this but it needs a rethink.

To do at some point

These are ideas that have been regularly tossed around, that most people believe should be done maybe during 5.8.x

Remove regular expression recursion

Because the regular expression engine is recursive, badly designed expressions can lead to lots of recursion filling up the stack. Ilya claims that it is easy to convert the engine to being iterative, but this has still not yet been done. There may be a regular expression engine hit squad meeting at TPC5.

Memory leaks after failed eval

Perl will leak memory if you eval "hlagh hlagh hlagh hlagh". This is partially because it attempts to build up an op tree for that code and doesn't properly free it. The same goes for non-syntactically-correct regular expressions. Hugo looked into this, but decided it needed a mark-and-sweep GC implementation.

Alan notes that: The basic idea was to extend the parser token stack (YYSTYPE) to include a type field so we knew what sort of thing each element of the stack was. The *perly.c* code would then have to be postprocessed to record the type of each entry on the stack as it was created, and the parser patched so that it could unroll the stack properly on error.

This is possible to do, but would be pretty messy to implement, as it would rely on even more sed hackery in *perlyfixer*.

bitfields in pack

Cross compilation

Make Perl buildable with a cross-compiler. This will play havoc with Configure, which needs to know how the target system will respond to its tests; maybe *microperl* will be a good starting point here. (Indeed, Bart Schuller reports that he compiled up *microperl* for the Agenda PDA and it works fine.) A really big spanner in the works is the bootstrapping build process of Perl: if the filesystem the target systems sees is not the same what the build host sees, various input, output, and (Perl) library files need to be copied back and forth.

As of 5.8.0 Configure mostly works for cross-compilation (used successfully for iPAQ Linux), *miniperl* gets built, but then building DynaLoader (and other extensions) fails since MakeMaker knows nothing of cross-compilation. (See INSTALL/Cross-compilation for the state of things.)

Perl preprocessor / macros

Source filters help with this, but do not get us all the way. For instance, it should be possible to implement the ?? operator somehow; source filters don't (quite) cut it.

Perl lexer in Perl

Damian Conway is planning to work on this, but it hasn't happened yet.

Using POSIX calls internally

When faced with a BSD vs. SysV -style interface to some library or system function, perl's roots show in that it typically prefers the BSD interface (but falls back to the SysV one). One example is `getpgrp()`. Other examples include `memcpy` vs. `bcopy`. There are others, mostly in *pp_sys.c*.

Mostly, this item is a suggestion for which way to start a journey into an `#ifdef` forest. It is not primarily a suggestion to eliminate any of the `#ifdef` forests.

POSIX calls are perhaps more likely to be portable to unexpected architectures. They are also perhaps more likely to be actively maintained by a current vendor. They are also perhaps more likely to be available in thread-safe versions, if appropriate.

-i rename file when changed

It's only necessary to rename a file when inplace editing when the file has changed. Detecting a change is perhaps the difficult bit.

All ARGV input should act like <>

eg `read(ARGV, ...)` doesn't currently read across multiple files.

Support for rerunning debugger

There should be a way of restarting the debugger on demand.

Test Suite for the Debugger

The debugger is a complex piece of software and fixing something here may inadvertently break something else over there. To tame this chaotic behaviour, a test suite is necessary.

my sub foo { }

The basic principle is sound, but there are problems with the semantics of self-referential and mutually referential lexical subs: how to declare the subs?

One-pass global destruction

Sweeping away all the allocated memory in one go is a laudable goal, but it's difficult and in most cases, it's easier to let the memory get freed by exiting.

Rewrite regexp parser

There has been talk recently of rewriting the regular expression parser to produce an optree instead of a chain of opcodes; it's unclear whether or not this would be a win.

Cache recently used regexps

This is to speed up

```
for my $re (@regexps) {
    $matched++ if /$re/
}
```

`qr//` already gives us a way of saving compiled regexps, but it should be done automatically.

Cross-compilation support

Bart Schuller reports that using `microperl` and a cross-compiler, he got Perl working on the Agenda PDA. However, one cannot build a full Perl because `Configure` needs to get the results for the target platform, for the host.

Bit-shifting bitvectors

Given:

```
vec($v, 1000, 1) = 1;
```

One should be able to do

```
$v <=<= 1;
```

and have the 999'th bit set.

Currently if you try with shift bitvectors you shift the NV/UV, instead of the bits in the PV. Not very logical.

debugger pragma

The debugger is implemented in Perl in *perl5db.pl*; turning it into a pragma should be easy, but making it work lexically might be more difficult. Fiddling with `$^P` would be necessary.

use less pragma

Identify areas where speed/memory tradeoffs can be made and have a hint to switch between them.

switch structures

Although we have `Switch.pm` in core, Larry points to the dormant `nswitch` and `cswitch` ops in *pp.c*; using these opcodes would be much faster.

Cache eval tree

rcatmaybe

Shrink opcode tables

Optimize away @_

Look at the "reification" code in `av.c`

Prototypes versus indirect objects

Currently, indirect object syntax bypasses prototype checks.

Install HTML

HTML versions of the documentation need to be installed by default; a call to `installhtml` from `installperl` may be all that's necessary.

Prototype method calls**Return context prototype declarations****`magic_setisa`****Garbage collection**

There have been persistent mumblings about putting a mark-and-sweep garbage detector into Perl; Alan Burlison has some ideas about this.

IO tutorial

Mark-Jason Dominus has the beginnings of one of these.

Rewrite `perldoc`

There are a few suggestions for what to do with `perldoc`: maybe a full-text search, an index function, locating pages on a particular high-level subject, and so on.

Install `.3p` manpages

This is a bone of contention; we can create `.3p` manpages for each built-in function, but should we install them by default? Tcl does this, and it clutters up `apropos`.

Unicode tutorial

Simon Cozens promises to do this before he gets old.

Update `POSIX.pm` for 1003.1-2**Retargetable installation**

Allow `@INC` to be changed after Perl is built.

POSIX emulation on non-POSIX systems

Make `POSIX.pm` behave as POSIXly as possible everywhere, meaning we have to implement POSIX equivalents for some functions if necessary.

Rename Win32 headers**Finish off `Ivalue` functions**

They don't work in the debugger, and they don't work for list or hash slices.

Update `sprintf` documentation

Hugo van der Sanden plans to look at this.

Use `fchown/fchmod` internally

This has been done in places, but needs a thorough code review. Also `fchdir` is available in some platforms.

Make `v-strings` overloaded objects

Instead of having to guess whether a string is a `v-string` and thus needs to be displayed with `%vd`, make `v-strings` (readonly) objects (class "vstring?") with a `stringify` overload.

Allow restricted hash assignment

Currently you're not allowed to assign to a restricted hash at all, even with the same keys.

```
%restricted = (foo => 42); # error
```

This should be allowed if the new keyset is a subset of the old keyset. May require more extra code than we'd like in pp_aassign.

Should overload be inheritable?

Should overload be 'contagious' through @ISA so that derived classes would inherit their base classes' overload definitions? What to do in case of overload conflicts?

Taint rethink

Should taint be stopped from affecting control flow, if (\$tainted)? Should tainted symbolic method calls and subref calls be stopped? (Look at Ruby's \$SAFE levels for inspiration?)

Vague ideas

Ideas which have been discussed, and which may or may not happen.

ref() in list context

It's unclear what this should do or how to do it without breaking old code.

Make tr/// return histogram of characters in list context

There is a patch for this, but it may require Unicodification.

Compile to real threaded code**Structured types****Modifiable \$! et al.**

```
($x = "elephant") =~ /e(ph)/;
$! = "g"; # $x = "elegant"
```

What happens if there are multiple (nested?) brackets? What if the string changes between the match and the assignment?

Procedural interfaces for IO::*, etc.

Some core modules have been accused of being overly-OO. Adding procedural interfaces could demystify them.

RPC modules**Attach/detach debugger from running program**

With gdb, you can attach the debugger to a running program if you pass the process ID. It would be good to do this with the Perl debugger on a running Perl program, although I'm not sure how it would be done.

GUI::Native

A non-core module that would use "native" GUI to create graphical applications.

foreach(reverse ...)

Currently

```
foreach (reverse @_) { ... }
```

puts @_ on the stack, reverses it putting the reversed version on the stack, then iterates forwards. Instead, it could be special-cased to put @_ on the stack then iterate backwards.

Constant function cache**Approximate regular expression matching**

Ongoing

These items **always** need doing:

Update guts documentation

Simon Cozens tries to do this when possible, and contributions to the `perlapi` documentation is welcome.

Add more tests

Michael Schwern will donate \$500 to Yet Another Society when all core modules have tests.

Update auxiliary tools

The code we ship with Perl should look like good Perl 5.

Create debugging macros

Debugging macros (like `printsv`, `dump`) can make debugging perl inside a C debugger much easier. A good set for `gdb` comes with `mod_perl`. Something similar should be distributed with perl.

The proper way to do this is to use and extend `Devel::DebugInit`. `Devel::DebugInit` also needs to be extended to support threads.

See p5p archives for late May/early June 2001 for a recent discussion on this topic.

truncate to the people

One can emulate `ftruncate()` using `F_FREESP` and `F_CHSIZ` `fcntl`s (see the UNIX FAQ for details). This needs to go somewhere near `pp_sys.c:pp_truncate()`.

One can emulate `truncate()` easily if one has `ftruncate()`. This emulation should also go near `pp_sys.pp_truncate()`.

Unicode in Filenames

`chdir`, `chmod`, `chown`, `chroot`, `exec`, `glob`, `link`, `lstat`, `mkdir`, `open`, `opendir`, `qx`, `readdir`, `readlink`, `rename`, `rmdir`, `stat`, `symlink`, `sysopen`, `system`, `truncate`, `unlink`, `utime`. All these could potentially accept Unicode filenames either as input or output (and in the case of `system` and `qx` Unicode in general, as input or output to/from the shell). Whether a filesystem – an operating system pair understands Unicode in filenames varies.

Known combinations that have some level of understanding include Microsoft NTFS, Apple HFS+ (In Mac OS 9 and X) and Apple UFS (in Mac OS X), NFS v4 is rumored to be Unicode, and of course Plan 9. How to create Unicode filenames, what forms of Unicode are accepted and used (UCS-2, UTF-16, UTF-8), what (if any) is the normalization form used, and so on, varies. Finding the right level of interfacing to Perl requires some thought. Remember that an OS does not implicate a filesystem.

Note that in Windows the `-C` command line flag already does quite a bit of the above (but even there the support is not complete: for example the `exec/spawn` are not Unicode-aware) by turning on the so-called "wide API support".

Recently done things

These are things which have been on the todo lists in previous releases but have recently been completed.

Alternative RE syntax module

The `Regexp::English` module, available from the CPAN, provides this:

```
my $re = Regexp::English
-> start_of_line
-> literal('Flippers')
-> literal(':')
-> optional
    -> whitespace_char
-> end
-> remember
    -> multiple
```

```
        -> digit;  
    /$re/;
```

Safe signal handling

A new signal model went into 5.7.1 without much fanfare. Operations and mallocs are no longer interrupted by signals, which are handled between opcodes. This means that `PERL_ASYNC_CHECK` now actually does something. However, there are still a few things that need to be done.

Tie Modules

Modules which implement arrays in terms of strings, substrings or files can be found on the CPAN.

gettimeofday

`Time::HiRes` has been integrated into the core.

setitimer and getitimer

Adding `Time::HiRes` got us this too.

Testing __DIE__ hook

Tests have been added.

CPP equivalent in Perl

A C Yardley will probably have done this by the time you can read this. This allows for a generalization of the C constant detection used in building `Errno.pm`.

Explicit switch statements

`Switch.pm` has been integrated into the core to give you all manner of `switch...case` semantics.

autocroak

This is `Fatal.pm`.

UTF/EBCDIC

Nick Ing-Simmons has made UTF-EBCDIC (UTR13) work with Perl.

EBCDIC? <http://www.unicode.org/unicode/reports/tr16/>

UTF Regexes

Although there are probably some small bugs to be rooted out, Jarkko Hietaniemi has made regular expressions polymorphic between bytes and characters.

perlcc to produce executable

`perlcc` was recently rewritten, and can now produce standalone executables.

END blocks saved in compiled output

Secure temporary file module

Tim Jenness' `File::Temp` is now in core.

Integrate Time::HiRes

This module is now part of core.

Turn Cwd into XS

Benjamin Sugars has done this.

Mmap for input

Nick Ing-Simmons' `perlio` supports an `mmap` IO method.

Byte to/from UTF8 and UTF8 to/from local conversion

`Encode` provides this.

Add socketmark support

Added in 5.7.1

Mailing list archives

<http://lists.perl.org/> , <http://archive.develooper.com/>

Bug tracking

Richard Foley has written the bug tracking system at <http://bugs.perl.org/>

Integrate MacPerl

Chris Nandor and Matthias Neeracher have integrated the MacPerl changes into 5.6.0.

Web "nerve center" for Perl

<http://use.perl.org/> is what you're looking for.

Regular expression tutorial

`perlretut`, provided by Mark Kvale.

Debugging Tutorial

`perldebtut`, written by Richard Foley.

Integrate new modules

Jarkko has been integrating madly into 5.7.x

Integrate profiler

`Devel::DProf` is now a core module.

Y2K error detection

There's a configure option to detect unsafe concatenation with "19", and a CPAN module. (`D'oh::Year`)

Regular expression debugger

While not part of core, Mark-Jason Dominus has written `Rx` and has also come up with a generalised strategy for regular expression debugging.

POD checker

That's, uh, *podchecker*

"Dynamic" lexicals**Cache precompiled modules****Deprecated Wishes**

These are items which used to be in the todo file, but have been deprecated for some reason.

Loop control on `do{}`

This would break old code; use `do { { } }` instead.

Lexically scoped `typoglobs`

Not needed now we have lexical IO handles.

format `BOTTOM`**report `HANDLE`**

Damian Conway's text formatting modules seem to be the Way To Go.

Generalised `want()`/`caller()`

Robin Houston's `Want` module does this.

Named prototypes

This seems to be delayed until Perl 6.

Built-in globbing

The `File::Glob` module has been used to replace the `glob` function.

Regression tests for `suidperl`

`suidperl` is deprecated in favour of common sense.

Cached hash values

We have shared hash keys, which perform the same job.

Add compression modules

The compression modules are a little heavy; meanwhile, Nick Clark is working on experimental pragmata to do transparent decompression on input.

Reorganise documentation into tutorials/references

Could not get consensus on P5P about this.

Remove distinction between functions and operators

Caution: highly flammable.

Make XS easier to use

Use `Inline` instead, or SWIG.

Make embedding easier to use

Use `Inline::CPR`.

man for perl

See the Perl Power Tools. (<http://language.perl.com/ppt/>)

`my $Package::variable`

Use `our` instead.

"or" tests defined, not truth

Suggesting this on P5P **will** cause a boring and interminable flamewar.

"class"-based lexicals

Use flyweight objects, secure hashes or, dare I say it, pseudo-hashes instead. (Or whatever will replace pseudohashes in 5.10.)

byteperl

`ByteLoader` covers this.

Lazy evaluation / tail recursion removal

`List::Util` gives `first()` (a short-circuiting `grep`); tail recursion removal is done manually, with `goto &whoami; .` (However, MJD has found that `goto &whoami` introduces a performance penalty, so maybe there should be a way to do this after all: `sub foo {START: ... goto START;}` is better.)

Make "use utf8" the default

Because of backward compatibility this is difficult: scripts could not contain **any legacy eight-bit data** (like Latin-1) anymore, even in string literals or pod. Also would introduce a measurable slowdown of at least few percentages since all regular expression operations would be done in full UTF-8. But if you want to try this, add `-DUSE_UTF8_SCRIPTS` to your compilation flags.

Unicode collation and normalization

The `Unicode::Collate` and `Unicode::Normalize` modules by SADAHIRO Tomoyuki have been included since 5.8.0.

Collation? <http://www.unicode.org/unicode/reports/tr10/>
Normalization? <http://www.unicode.org/unicode/reports/tr15/>

pack/unpack tutorial

Wolfgang Laun finished what Simon Cozens started.

NAME

perltooc – Tom’s OO Tutorial for Class Data in Perl

DESCRIPTION

When designing an object class, you are sometimes faced with the situation of wanting common state shared by all objects of that class. Such *class attributes* act somewhat like global variables for the entire class, but unlike program-wide globals, class attributes have meaning only to the class itself.

Here are a few examples where class attributes might come in handy:

- to keep a count of the objects you’ve created, or how many are still extant.
- to extract the name or file descriptor for a logfile used by a debugging method.
- to access collective data, like the total amount of cash dispensed by all ATMs in a network in a given day.
- to access the last object created by a class, or the most accessed object, or to retrieve a list of all objects.

Unlike a true global, class attributes should not be accessed directly. Instead, their state should be inspected, and perhaps altered, only through the mediated access of *class methods*. These class attributes accessor methods are similar in spirit and function to accessors used to manipulate the state of instance attributes on an object. They provide a clear firewall between interface and implementation.

You should allow access to class attributes through either the class name or any object of that class. If we assume that `$an_object` is of type `Some_Class`, and the `&Some_Class::population_count` method accesses class attributes, then these two invocations should both be possible, and almost certainly equivalent.

```
Some_Class->population_count()  
$an_object->population_count()
```

The question is, where do you store the state which that method accesses? Unlike more restrictive languages like C++, where these are called static data members, Perl provides no syntactic mechanism to declare class attributes, any more than it provides a syntactic mechanism to declare instance attributes. Perl provides the developer with a broad set of powerful but flexible features that can be uniquely crafted to the particular demands of the situation.

A class in Perl is typically implemented in a module. A module consists of two complementary feature sets: a package for interfacing with the outside world, and a lexical file scope for privacy. Either of these two mechanisms can be used to implement class attributes. That means you get to decide whether to put your class attributes in package variables or to put them in lexical variables.

And those aren’t the only decisions to make. If you choose to use package variables, you can make your class attribute accessor methods either ignorant of inheritance or sensitive to it. If you choose lexical variables, you can elect to permit access to them from anywhere in the entire file scope, or you can limit direct data access exclusively to the methods implementing those attributes.

Class Data in a Can

One of the easiest ways to solve a hard problem is to let someone else do it for you! In this case, `Class::Data::Inheritable` (available on a CPAN near you) offers a canned solution to the class data problem using closures. So before you waded into this document, consider having a look at that module.

Class Data as Package Variables

Because a class in Perl is really just a package, using package variables to hold class attributes is the most natural choice. This makes it simple for each class to have its own class attributes. Let’s say you have a class called `Some_Class` that needs a couple of different attributes that you’d like to be global to the entire class. The simplest thing to do is to use package variables like `$Some_Class::CData1` and `$Some_Class::CData2` to hold these attributes. But we certainly don’t want to encourage outsiders to

touch those data directly, so we provide methods to mediate access.

In the accessor methods below, we'll for now just ignore the first argument—that part to the left of the arrow on method invocation, which is either a class name or an object reference.

```
package Some_Class;
sub CData1 {
    shift; # XXX: ignore calling class/object
    $Some_Class::CData1 = shift if @_;
    return $Some_Class::CData1;
}
sub CData2 {
    shift; # XXX: ignore calling class/object
    $Some_Class::CData2 = shift if @_;
    return $Some_Class::CData2;
}
```

This technique is highly legible and should be completely straightforward to even the novice Perl programmer. By fully qualifying the package variables, they stand out clearly when reading the code. Unfortunately, if you misspell one of these, you've introduced an error that's hard to catch. It's also somewhat disconcerting to see the class name itself hard-coded in so many places.

Both these problems can be easily fixed. Just add the `use strict` pragma, then pre-declare your package variables. (The `our` operator will be new in 5.6, and will work for package globals just like `my` works for scoped lexicals.)

```
package Some_Class;
use strict;
our($CData1, $CData2); # our() is new to perl5.6
sub CData1 {
    shift; # XXX: ignore calling class/object
    $CData1 = shift if @_;
    return $CData1;
}
sub CData2 {
    shift; # XXX: ignore calling class/object
    $CData2 = shift if @_;
    return $CData2;
}
```

As with any other global variable, some programmers prefer to start their package variables with capital letters. This helps clarity somewhat, but by no longer fully qualifying the package variables, their significance can be lost when reading the code. You can fix this easily enough by choosing better names than were used here.

Putting All Your Eggs in One Basket

Just as the mindless enumeration of accessor methods for instance attributes grows tedious after the first few (see [perltoot](#)), so too does the repetition begin to grate when listing out accessor methods for class data. Repetition runs counter to the primary virtue of a programmer: Laziness, here manifesting as that innate urge every programmer feels to factor out duplicate code whenever possible.

Here's what to do. First, make just one hash to hold all class attributes.

```
package Some_Class;
use strict;
our %ClassData = ( # our() is new to perl5.6
    CData1 => "",
    CData2 => "",
);
```

Using closures (see [perlref](#)) and direct access to the package symbol table (see [perlmod](#)), now clone an accessor method for each key in the %ClassData hash. Each of these methods is used to fetch or store values to the specific, named class attribute.

```
for my $datum (keys %ClassData) {
    no strict "refs";          # to register new methods in package
    *$datum = sub {
        shift;                # XXX: ignore calling class/object
        $ClassData{$datum} = shift if @_ ;
        return $ClassData{$datum};
    }
}
```

It's true that you could work out a solution employing an &AUTOLOAD method, but this approach is unlikely to prove satisfactory. Your function would have to distinguish between class attributes and object attributes; it could interfere with inheritance; and it would have to be careful about DESTROY. Such complexity is uncalled for in most cases, and certainly in this one.

You may wonder why we're rescinding strict refs for the loop. We're manipulating the package's symbol table to introduce new function names using symbolic references (indirect naming), which the strict pragma would otherwise forbid. Normally, symbolic references are a dodgy notion at best. This isn't just because they can be used accidentally when you aren't meaning to. It's also because for most uses to which beginning Perl programmers attempt to put symbolic references, we have much better approaches, like nested hashes or hashes of arrays. But there's nothing wrong with using symbolic references to manipulate something that is meaningful only from the perspective of the package symbol table, like method names or package variables. In other words, when you want to refer to the symbol table, use symbol references.

Clustering all the class attributes in one place has several advantages. They're easy to spot, initialize, and change. The aggregation also makes them convenient to access externally, such as from a debugger or a persistence package. The only possible problem is that we don't automatically know the name of each class's class object, should it have one. This issue is addressed below in *"The Eponymous Meta-Object"*.

Inheritance Concerns

Suppose you have an instance of a derived class, and you access class data using an inherited method call. Should that end up referring to the base class's attributes, or to those in the derived class? How would it work in the earlier examples? The derived class inherits all the base class's methods, including those that access class attributes. But what package are the class attributes stored in?

The answer is that, as written, class attributes are stored in the package into which those methods were compiled. When you invoke the &CData1 method on the name of the derived class or on one of that class's objects, the version shown above is still run, so you'll access \$Some_Class::CData1—or in the method cloning version, \$Some_Class::ClassData{CData1}.

Think of these class methods as executing in the context of their base class, not in that of their derived class. Sometimes this is exactly what you want. If Feline subclasses Carnivore, then the population of Carnivores in the world should go up when a new Feline is born. But what if you wanted to figure out how many Felines you have apart from Carnivores? The current approach doesn't support that.

You'll have to decide on a case-by-case basis whether it makes any sense for class attributes to be package-relative. If you want it to be so, then stop ignoring the first argument to the function. Either it will be a package name if the method was invoked directly on a class name, or else it will be an object reference if the method was invoked on an object reference. In the latter case, the `ref()` function provides the class of that object.

```
package Some_Class;
sub CData1 {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $varname = $class . "::CData1";
```

```

        no strict "refs" # to access package data symbolically
        $$varname = shift if @_;
        return $$varname;
    }

```

And then do likewise for all other class attributes (such as CData2, etc.) that you wish to access as package variables in the invoking package instead of the compiling package as we had previously.

Once again we temporarily disable the strict references ban, because otherwise we couldn't use the fully-qualified symbolic name for the package global. This is perfectly reasonable: since all package variables by definition live in a package, there's nothing wrong with accessing them via that package's symbol table. That's what it's there for (well, somewhat).

What about just using a single hash for everything and then cloning methods? What would that look like? The only difference would be the closure used to produce new method entries for the class's symbol table.

```

no strict "refs";
*$datum = sub {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $varname = $class . "::ClassData";
    $varname->{$datum} = shift if @_;
    return $varname->{$datum};
}

```

The Eponymous Meta-Object

It could be argued that the %ClassData hash in the previous example is neither the most imaginative nor the most intuitive of names. Is there something else that might make more sense, be more useful, or both?

As it happens, yes, there is. For the "class meta-object", we'll use a package variable of the same name as the package itself. Within the scope of a package `Some_Class` declaration, we'll use the eponymously named hash %Some_Class as that class's meta-object. (Using an eponymously named hash is somewhat reminiscent of classes that name their constructors eponymously in the Python or C++ fashion. That is, class `Some_Class` would use `&Some_Class::Some_Class` as a constructor, probably even exporting that name as well. The `StrNum` class in Recipe 13.14 in *The Perl Cookbook* does this, if you're looking for an example.)

This predictable approach has many benefits, including having a well-known identifier to aid in debugging, transparent persistence, or checkpointing. It's also the obvious name for monadic classes and translucent attributes, discussed later.

Here's an example of such a class. Notice how the name of the hash storing the meta-object is the same as the name of the package used to implement the class.

```

package Some_Class;
use strict;

# create class meta-object using that most perfect of names
our %Some_Class = (          # our() is new to perl5.6
    CData1 => "",
    CData2 => "",
);

# this accessor is calling-package-relative
sub CData1 {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    no strict "refs";          # to access eponymous meta-object
    $class->{CData1} = shift if @_;
    return $class->{CData1};
}

```

```

}

# but this accessor is not
sub CData2 {
    shift;                      # XXX: ignore calling class/object
    no strict "refs";           # to access eponymous meta-object
    __PACKAGE__ -> {CData2} = shift if @_;
    return __PACKAGE__ -> {CData2};
}

```

In the second accessor method, the `__PACKAGE__` notation was used for two reasons. First, to avoid hardcoding the literal package name in the code in case we later want to change that name. Second, to clarify to the reader that what matters here is the package currently being compiled into, not the package of the invoking object or class. If the long sequence of non-alphabetic characters bothers you, you can always put the `__PACKAGE__` in a variable first.

```

sub CData2 {
    shift;                      # XXX: ignore calling class/object
    no strict "refs";           # to access eponymous meta-object
    my $class = __PACKAGE__;
    $class->{CData2} = shift if @_;
    return $class->{CData2};
}

```

Even though we're using symbolic references for good not evil, some folks tend to become unnerved when they see so many places with strict ref checking disabled. Given a symbolic reference, you can always produce a real reference (the reverse is not true, though). So we'll create a subroutine that does this conversion for us. If invoked as a function of no arguments, it returns a reference to the compiling class's eponymous hash. Invoked as a class method, it returns a reference to the eponymous hash of its caller. And when invoked as an object method, this function returns a reference to the eponymous hash for whatever class the object belongs to.

```

package Some_Class;
use strict;

our %Some_Class = (           # our() is new to perl5.6
    CData1 => "",
    CData2 => "",
);

# tri-natured: function, class method, or object method
sub _classobj {
    my $obclass = shift || __PACKAGE__;
    my $class   = ref($obclass) || $obclass;
    no strict "refs"; # to convert sym ref to real one
    return \%$class;
}

for my $datum (keys %{ _classobj() } ) {
    # turn off strict refs so that we can
    # register a method in the symbol table
    no strict "refs";
    *$datum = sub {
        use strict "refs";
        my $self = shift->_classobj();
        $self->{$datum} = shift if @_;
        return $self->{$datum};
    }
}

```



```
}
```

Indirect References to Class Data

A reasonably common strategy for handling class attributes is to store a reference to each package variable on the object itself. This is a strategy you've probably seen before, such as in [perltoot](#) and [perlbot](#), but there may be variations in the example below that you haven't thought of before.

```
package Some_Class;
our($CData1, $CData2);          # our() is new to perl5.6

sub new {
    my $obclass = shift;
    return bless my $self = {
        ObData1 => "",
        ObData2 => "",
        CData1  => \$CData1,
        CData2  => \$CData2,
    } => (ref $obclass || $obclass);
}

sub ObData1 {
    my $self = shift;
    $self->{ObData1} = shift if @_;
    return $self->{ObData1};
}

sub ObData2 {
    my $self = shift;
    $self->{ObData2} = shift if @_;
    return $self->{ObData2};
}

sub CData1 {
    my $self = shift;
    my $dataref = ref $self
        ? $self->{CData1}
        : \$CData1;
    $$dataref = shift if @_;
    return $$dataref;
}

sub CData2 {
    my $self = shift;
    my $dataref = ref $self
        ? $self->{CData2}
        : \$CData2;
    $$dataref = shift if @_;
    return $$dataref;
}
```

As written above, a derived class will inherit these methods, which will consequently access package variables in the base class's package. This is not necessarily expected behavior in all circumstances. Here's an example that uses a variable meta-object, taking care to access the proper package's data.

```
package Some_Class;
use strict;

our %Some_Class = (          # our() is new to perl5.6
    CData1 => "",
```

```

        CData2 => "",
    );
    sub _classobj {
        my $self = shift;
        my $class = ref($self) || $self;
        no strict "refs";
        # get (hard) ref to eponymous meta-object
        return \%$class;
    }
    sub new {
        my $obclass = shift;
        my $classobj = $obclass->_classobj();
        bless my $self = {
            ObData1 => "",
            ObData2 => "",
            CData1 => \%$classobj->{CData1},
            CData2 => \%$classobj->{CData2},
        } => (ref $obclass || $obclass);
        return $self;
    }
    sub ObData1 {
        my $self = shift;
        $self->{ObData1} = shift if @_;
        return $self->{ObData1};
    }
    sub ObData2 {
        my $self = shift;
        $self->{ObData2} = shift if @_;
        return $self->{ObData2};
    }
    sub CData1 {
        my $self = shift;
        $self = $self->_classobj() unless ref $self;
        my $dataref = $self->{CData1};
        $$dataref = shift if @_;
        return $$dataref;
    }
    sub CData2 {
        my $self = shift;
        $self = $self->_classobj() unless ref $self;
        my $dataref = $self->{CData2};
        $$dataref = shift if @_;
        return $$dataref;
    }
}

```

Not only are we now strict refs clean, using an eponymous meta-object seems to make the code cleaner. Unlike the previous version, this one does something interesting in the face of inheritance: it accesses the class meta-object in the invoking class instead of the one into which the method was initially compiled.

You can easily access data in the class meta-object, making it easy to dump the complete class state using an external mechanism such as when debugging or implementing a persistent class. This works because the class meta-object is a package variable, has a well-known name, and clusters all its data together.

(Transparent persistence is not always feasible, but it's certainly an appealing idea.)

There's still no check that object accessor methods have not been invoked on a class name. If strict ref checking is enabled, you'd blow up. If not, then you get the eponymous meta-object. What you do with—or about—this is up to you. The next two sections demonstrate innovative uses for this powerful feature.

Monadic Classes

Some of the standard modules shipped with Perl provide class interfaces without any attribute methods whatsoever. The most commonly used module not numbered amongst the pragmata, the `Exporter` module, is a class with neither constructors nor attributes. Its job is simply to provide a standard interface for modules wishing to export part of their namespace into that of their caller. Modules use the `Exporter`'s `&import` method by setting their inheritance list in their package's `@ISA` array to mention "Exporter". But class `Exporter` provides no constructor, so you can't have several instances of the class. In fact, you can't have any—it just doesn't make any sense. All you get is its methods. Its interface contains no statefulness, so state data is wholly superfluous.

Another sort of class that pops up from time to time is one that supports a unique instance. Such classes are called *monadic classes*, or less formally, *singletons* or *highlander classes*.

If a class is monadic, where do you store its state, that is, its attributes? How do you make sure that there's never more than one instance? While you could merely use a slew of package variables, it's a lot cleaner to use the eponymously named hash. Here's a complete example of a monadic class:

```
package Cosmos;
%Cosmos = ();

# accessor method for "name" attribute
sub name {
    my $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}

# read-only accessor method for "birthday" attribute
sub birthday {
    my $self = shift;
    die "can't reset birthday" if @_; # XXX: croak() is better
    return $self->{birthday};
}

# accessor method for "stars" attribute
sub stars {
    my $self = shift;
    $self->{stars} = shift if @_;
    return $self->{stars};
}

# oh my - one of our stars just went out!
sub supernova {
    my $self = shift;
    my $count = $self->stars();
    $self->stars($count - 1) if $count > 0;
}

# constructor/initializer method - fix by reboot
sub bigbang {
    my $self = shift;
    %$self = (
```

```

        name => "the world according to tchrist",
        birthday    => time(),
        stars=> 0,
    );
    return $self;          # yes, it's probably a class.  SURPRISE!
}

# After the class is compiled, but before any use or require
# returns, we start off the universe with a bang.
__PACKAGE__ -> bigbang();

```

Hold on, that doesn't look like anything special. Those attribute accessors look no different than they would if this were a regular class instead of a monadic one. The crux of the matter is there's nothing that says that `$self` must hold a reference to a blessed object. It merely has to be something you can invoke methods on.

Here the package name itself, `Cosmos`, works as an object. Look at the `&supernova` method. Is that a class method or an object method? The answer is that static analysis cannot reveal the answer. Perl doesn't care, and neither should you. In the three attribute methods, `$self` is really accessing the `%Cosmos` package variable.

If like Stephen Hawking, you posit the existence of multiple, sequential, and unrelated universes, then you can invoke the `&bigbang` method yourself at any time to start everything all over again. You might think of `&bigbang` as more of an initializer than a constructor, since the function doesn't allocate new memory; it only initializes what's already there. But like any other constructor, it does return a scalar value to use for later method invocations.

Imagine that some day in the future, you decide that one universe just isn't enough. You could write a new class from scratch, but you already have an existing class that does what you want—except that it's monadic, and you want more than just one cosmos.

That's what code reuse via subclassing is all about. Look how short the new code is:

```

package Multiverse;
use Cosmos;
@ISA = qw(Cosmos);

sub new {
    my $protoverse = shift;
    my $class      = ref($protoverse) || $protoverse;
    my $self       = {};
    return bless($self, $class)->bigbang();
}
1;

```

Because we were careful to be good little creators when we designed our `Cosmos` class, we can now reuse it without touching a single line of code when it comes time to write our `Multiverse` class. The same code that worked when invoked as a class method continues to work perfectly well when invoked against separate instances of a derived class.

The astonishing thing about the `Cosmos` class above is that the value returned by the `&bigbang` "constructor" is not a reference to a blessed object at all. It's just the class's own name. A class name is, for virtually all intents and purposes, a perfectly acceptable object. It has state, behavior, and identify, the three crucial components of an object system. It even manifests inheritance, polymorphism, and encapsulation. And what more can you ask of an object?

To understand object orientation in Perl, it's important to recognize the unification of what other programming languages might think of as class methods and object methods into just plain methods. "Class methods" and "object methods" are distinct only in the compartmentalizing mind of the Perl programmer, not in the Perl language itself.

Along those same lines, a constructor is nothing special either, which is one reason why Perl has no pre-ordained name for them. "Constructor" is just an informal term loosely used to describe a method that returns a scalar value that you can make further method calls against. So long as it's either a class name or an object reference, that's good enough. It doesn't even have to be a reference to a brand new object.

You can have as many—or as few—constructors as you want, and you can name them whatever you care to. Blindly and obediently using `new()` for each and every constructor you ever write is to speak Perl with such a severe C++ accent that you do a disservice to both languages. There's no reason to insist that each class have but one constructor, or that a constructor be named `new()`, or that a constructor be used solely as a class method and not an object method.

The next section shows how useful it can be to further distance ourselves from any formal distinction between class method calls and object method calls, both in constructors and in accessor methods.

Translucent Attributes

A package's eponymous hash can be used for more than just containing per-class, global state data. It can also serve as a sort of template containing default settings for object attributes. These default settings can then be used in constructors for initialization of a particular object. The class's eponymous hash can also be used to implement *translucent attributes*. A translucent attribute is one that has a class-wide default. Each object can set its own value for the attribute, in which case `< $object->attribute()` returns that value. But if no value has been set, then `< $object->attribute()` returns the class-wide default.

We'll apply something of a copy-on-write approach to these translucent attributes. If you're just fetching values from them, you get translucency. But if you store a new value to them, that new value is set on the current object. On the other hand, if you use the class as an object and store the attribute value directly on the class, then the meta-object's value changes, and later fetch operations on objects with uninitialized values for those attributes will retrieve the meta-object's new values. Objects with their own initialized values, however, won't see any change.

Let's look at some concrete examples of using these properties before we show how to implement them. Suppose that a class named `Some_Class` had a translucent data attribute called "color". First you set the color in the meta-object, then you create three objects using a constructor that happens to be named `&spawn`.

```
use Vermin;
Vermin->color("vermilion");

$ob1 = Vermin->spawn();      # so that's where Jedi come from
$ob2 = Vermin->spawn();
$ob3 = Vermin->spawn();

print $obj3->color();        # prints "vermilion"
```

Each of these objects' colors is now "vermilion", because that's the meta-object's value that attribute, and these objects do not have individual color values set.

Changing the attribute on one object has no effect on other objects previously created.

```
$ob3->color("chartreuse");
print $ob3->color();          # prints "chartreuse"
print $ob1->color();          # prints "vermilion", translucently
```

If you now use `$ob3` to spawn off another object, the new object will take the color its parent held, which now happens to be "chartreuse". That's because the constructor uses the invoking object as its template for initializing attributes. When that invoking object is the class name, the object used as a template is the eponymous meta-object. When the invoking object is a reference to an instantiated object, the `&spawn` constructor uses that existing object as a template.

```
$ob4 = $ob3->spawn();          # $ob3 now template, not %Vermin
print $ob4->color();           # prints "chartreuse"
```

Any actual values set on the template object will be copied to the new object. But attributes undefined in the template object, being translucent, will remain undefined and consequently translucent in the new one as well.

Now let's change the color attribute on the entire class:

```
Vermin->color("azure");
print $ob1->color();      # prints "azure"
print $ob2->color();      # prints "azure"
print $ob3->color();      # prints "chartreuse"
print $ob4->color();      # prints "chartreuse"
```

That color change took effect only in the first pair of objects, which were still translucently accessing the meta-object's values. The second pair had per-object initialized colors, and so didn't change.

One important question remains. Changes to the meta-object are reflected in translucent attributes in the entire class, but what about changes to discrete objects? If you change the color of \$ob3, does the value of \$ob4 see that change? Or vice-versa. If you change the color of \$ob4, does then the value of \$ob3 shift?

```
$ob3->color("amethyst");
print $ob3->color();      # prints "amethyst"
print $ob4->color();      # hmm: "chartreuse" or "amethyst"?
```

While one could argue that in certain rare cases it should, let's not do that. Good taste aside, we want the answer to the question posed in the comment above to be "chartreuse", not "amethyst". So we'll treat these attributes similar to the way process attributes like environment variables, user and group IDs, or the current working directory are treated across a `fork()`. You can change only yourself, but you will see those changes reflected in your unspawned children. Changes to one object will propagate neither up to the parent nor down to any existing child objects. Those objects made later, however, will see the changes.

If you have an object with an actual attribute value, and you want to make that object's attribute value translucent again, what do you do? Let's design the class so that when you invoke an accessor method with `undef` as its argument, that attribute returns to translucency.

```
$ob4->color(undef);      # back to "azure"
```

Here's a complete implementation of Vermin as described above.

```
package Vermin;

# here's the class meta-object, eponymously named.
# it holds all class attributes, and also all instance attributes
# so the latter can be used for both initialization
# and translucency.

our %Vermin = (          # our() is new to perl5.6
    PopCount => 0,        # capital for class attributes
    color    => "beige",  # small for instance attributes
);

# constructor method
# invoked as class method or object method
sub spawn {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $self = {};
    bless($self, $class);
    $class->{PopCount}++;
    # init fields from invoking object, or omit if
    # invoking object is the class to provide translucency
```

```

        %$self = %$obclass if ref $obclass;
        return $self;
    }

    # translucent accessor for "color" attribute
    # invoked as class method or object method
    sub color {
        my $self = shift;
        my $class = ref($self) || $self;

        # handle class invocation
        unless (ref $self) {
            $class->{color} = shift if @_;
            return $class->{color}
        }

        # handle object invocation
        $self->{color} = shift if @_;
        if (defined $self->{color}) { # not exists!
            return $self->{color};
        } else {
            return $class->{color};
        }
    }

    # accessor for "PopCount" class attribute
    # invoked as class method or object method
    # but uses object solely to locate meta-object
    sub population {
        my $obclass = shift;
        my $class = ref($obclass) || $obclass;
        return $class->{PopCount};
    }

    # instance destructor
    # invoked only as object method
    sub DESTROY {
        my $self = shift;
        my $class = ref $self;
        $class->{PopCount}--;
    }

```

Here are a couple of helper methods that might be convenient. They aren't accessor methods at all. They're used to detect accessibility of data attributes. The `&is_translucent` method determines whether a particular object attribute is coming from the meta-object. The `&has_attribute` method detects whether a class implements a particular property at all. It could also be used to distinguish undefined properties from non-existent ones.

```

    # detect whether an object attribute is translucent
    # (typically?) invoked only as object method
    sub is_translucent {
        my($self, $attr) = @_;
        return !defined $self->{$attr};
    }

    # test for presence of attribute in class
    # invoked as class method or object method
    sub has_attribute {

```

```

    my($self, $attr) = @_;
    my $class = ref $self if $self;
    return exists $class->{$attr};
}

```

If you prefer to install your accessors more generically, you can make use of the upper-case versus lower-case convention to register into the package appropriate methods cloned from generic closures.

```

for my $datum (keys %{ +__PACKAGE__ }) {
    *$datum = ($datum =~ /^[A-Z]/)
        ? sub { # install class accessor
            my $obclass = shift;
            my $class = ref($obclass) || $obclass;
            return $class->{$datum};
        }
        : sub { # install translucent accessor
            my $self = shift;
            my $class = ref($self) || $self;
            unless (ref $self) {
                $class->{$datum} = shift if @_;
                return $class->{$datum}
            }
            $self->{$datum} = shift if @_;
            return defined $self->{$datum}
                ? $self -> {$datum}
                : $class -> {$datum}
        }
}

```

Translations of this closure-based approach into C++, Java, and Python have been left as exercises for the reader. Be sure to send us mail as soon as you're done.

Class Data as Lexical Variables

Privacy and Responsibility

Unlike conventions used by some Perl programmers, in the previous examples, we didn't prefix the package variables used for class attributes with an underscore, nor did we do so for the names of the hash keys used for instance attributes. You don't need little markers on data names to suggest nominal privacy on attribute variables or hash keys, because these are **already** notionally private! Outsiders have no business whatsoever playing with anything within a class save through the mediated access of its documented interface; in other words, through method invocations. And not even through just any method, either. Methods that begin with an underscore are traditionally considered off-limits outside the class. If outsiders skip the documented method interface to poke around the internals of your class and end up breaking something, that's not your fault—it's theirs.

Perl believes in individual responsibility rather than mandated control. Perl respects you enough to let you choose your own preferred level of pain, or of pleasure. Perl believes that you are creative, intelligent, and capable of making your own decisions—and fully expects you to take complete responsibility for your own actions. In a perfect world, these admonitions alone would suffice, and everyone would be intelligent, responsible, happy, and creative. And careful. One probably shouldn't forget careful, and that's a good bit harder to expect. Even Einstein would take wrong turns by accident and end up lost in the wrong part of town.

Some folks get the heebie-jeebies when they see package variables hanging out there for anyone to reach over and alter them. Some folks live in constant fear that someone somewhere might do something wicked. The solution to that problem is simply to fire the wicked, of course. But unfortunately, it's not as simple as all that. These cautious types are also afraid that they or others will do something not so much wicked as careless, whether by accident or out of desperation. If we fire everyone who ever gets careless, pretty soon

there won't be anybody left to get any work done.

Whether it's needless paranoia or sensible caution, this uneasiness can be a problem for some people. We can take the edge off their discomfort by providing the option of storing class attributes as lexical variables instead of as package variables. The `my()` operator is the source of all privacy in Perl, and it is a powerful form of privacy indeed.

It is widely perceived, and indeed has often been written, that Perl provides no data hiding, that it affords the class designer no privacy nor isolation, merely a rag-tag assortment of weak and unenforcible social conventions instead. This perception is demonstrably false and easily disproven. In the next section, we show how to implement forms of privacy that are far stronger than those provided in nearly any other object-oriented language.

File-Scoped Lexicals

A lexical variable is visible only through the end of its static scope. That means that the only code able to access that variable is code residing textually below the `my()` operator through the end of its block if it has one, or through the end of the current file if it doesn't.

Starting again with our simplest example given at the start of this document, we replace `our()` variables with `my()` versions.

```
package Some_Class;
my($CData1, $CData2);    # file scope, not in any package
sub CData1 {
    shift; # XXX: ignore calling class/object
    $CData1 = shift if @_;
    return $CData1;
}
sub CData2 {
    shift; # XXX: ignore calling class/object
    $CData2 = shift if @_;
    return $CData2;
}
```

So much for that old `$Some_Class::CData1` package variable and its brethren! Those are gone now, replaced with lexicals. No one outside the scope can reach in and alter the class state without resorting to the documented interface. Not even subclasses or superclasses of this one have unmediated access to `$CData1`. They have to invoke the `&CData1` method against `Some_Class` or an instance thereof, just like anybody else.

To be scrupulously honest, that last statement assumes you haven't packed several classes together into the same file scope, nor strewn your class implementation across several different files. Accessibility of those variables is based uniquely on the static file scope. It has nothing to do with the package. That means that code in a different file but the same package (class) could not access those variables, yet code in the same file but a different package (class) could. There are sound reasons why we usually suggest a one-to-one mapping between files and packages and modules and classes. You don't have to stick to this suggestion if you really know what you're doing, but you're apt to confuse yourself otherwise, especially at first.

If you'd like to aggregate your class attributes into one lexically scoped, composite structure, you're perfectly free to do so.

```
package Some_Class;
my %ClassData = (
    CData1 => "",
    CData2 => "",
);
sub CData1 {
    shift; # XXX: ignore calling class/object
    $ClassData{CData1} = shift if @_;
}
```

```

        return $ClassData{CData1};
    }
    sub CData2 {
        shift; # XXX: ignore calling class/object
        $ClassData{CData2} = shift if @_;
        return $ClassData{CData2};
    }

```

To make this more scalable as other class attributes are added, we can again register closures into the package symbol table to create accessor methods for them.

```

package Some_Class;
my %ClassData = (
    CData1 => "",
    CData2 => "",
);
for my $datum (keys %ClassData) {
    no strict "refs";
    *$datum = sub {
        shift; # XXX: ignore calling class/object
        $ClassData{$datum} = shift if @_;
        return $ClassData{$datum};
    };
}

```

Requiring even your own class to use accessor methods like anybody else is probably a good thing. But demanding and expecting that everyone else, be they subclass or superclass, friend or foe, will all come to your object through mediation is more than just a good idea. It's absolutely critical to the model. Let there be in your mind no such thing as "public" data, nor even "protected" data, which is a seductive but ultimately destructive notion. Both will come back to bite at you. That's because as soon as you take that first step out of the solid position in which all state is considered completely private, save from the perspective of its own accessor methods, you have violated the envelope. And, having pierced that encapsulating envelope, you shall doubtless someday pay the price when future changes in the implementation break unrelated code. Considering that avoiding this infelicitous outcome was precisely why you consented to suffer the slings and arrows of obsequious abstraction by turning to object orientation in the first place, such breakage seems unfortunate in the extreme.

More Inheritance Concerns

Suppose that `Some_Class` were used as a base class from which to derive `Another_Class`. If you invoke a `&CData` method on the derived class or on an object of that class, what do you get? Would the derived class have its own state, or would it piggyback on its base class's versions of the class attributes?

The answer is that under the scheme outlined above, the derived class would **not** have its own state data. As before, whether you consider this a good thing or a bad one depends on the semantics of the classes involved.

The cleanest, sanest, simplest way to address per-class state in a lexical is for the derived class to override its base class's version of the method that accesses the class attributes. Since the actual method called is the one in the object's derived class if this exists, you automatically get per-class state this way. Any urge to provide an unadvertised method to sneak out a reference to the `%ClassData` hash should be strenuously resisted.

As with any other overridden method, the implementation in the derived class always has the option of invoking its base class's version of the method in addition to its own. Here's an example:

```

package Another_Class;
@ISA = qw(Some_Class);

```

```

my %ClassData = (
    CData1 => "",
);

sub CData1 {
    my($self, $newvalue) = @_;
    if (@_ > 1) {
        # set locally first
        $ClassData{CData1} = $newvalue;

        # then pass the buck up to the first
        # overridden version, if there is one
        if ($self->can("SUPER::CData1")) {
            $self->SUPER::CData1($newvalue);
        }
    }
    return $ClassData{CData1};
}

```

Those dabbling in multiple inheritance might be concerned about there being more than one override.

```

for my $parent (@ISA) {
    my $methname = $parent . "::CData1";
    if ($self->can($methname)) {
        $self->$methname($newvalue);
    }
}

```

Because the `&UNIVERSAL::can` method returns a reference to the function directly, you can use this directly for a significant performance improvement:

```

for my $parent (@ISA) {
    if (my $coderef = $self->can($parent . "::CData1")) {
        $self->$coderef($newvalue);
    }
}

```

Locking the Door and Throwing Away the Key

As currently implemented, any code within the same scope as the file-scoped lexical `%ClassData` can alter that hash directly. Is that ok? Is it acceptable or even desirable to allow other parts of the implementation of this class to access class attributes directly?

That depends on how careful you want to be. Think back to the `Cosmos` class. If the `&supernova` method had directly altered `$Cosmos::Stars` or `$Cosmos::Cosmos{stars}`, then we wouldn't have been able to reuse the class when it came to inventing a `Multiverse`. So letting even the class itself access its own class attributes without the mediating intervention of properly designed accessor methods is probably not a good idea after all.

Restricting access to class attributes from the class itself is usually not enforceable even in strongly object-oriented languages. But in Perl, you can.

Here's one way:

```

package Some_Class;

{ # scope for hiding $CData1
    my $CData1;
    sub CData1 {
        shift; # XXX: unused
        $CData1 = shift if @_;
    }
}

```

```

        return $CData1;
    }
}
{ # scope for hiding $CData2
    my $CData2;
    sub CData2 {
        shift;      # XXX: unused
        $CData2 = shift if @_;
        return $CData2;
    }
}

```

No one—absolutely no one—is allowed to read or write the class attributes without the mediation of the managing accessor method, since only that method has access to the lexical variable it's managing. This use of mediated access to class attributes is a form of privacy far stronger than most OO languages provide.

The repetition of code used to create per-datum accessor methods chafes at our Laziness, so we'll again use closures to create similar methods.

```

package Some_Class;

{ # scope for ultra-private meta-object for class attributes
    my %ClassData = (
        CData1 => "",
        CData2 => "",
    );

    for my $datum (keys %ClassData) {
        no strict "refs";
        *$datum = sub {
            use strict "refs";
            my ($self, $newvalue) = @_;
            $ClassData{$datum} = $newvalue if @_ > 1;
            return $ClassData{$datum};
        }
    }
}

```

The closure above can be modified to take inheritance into account using the `&UNIVERSAL::can` method and `SUPER` as shown previously.

Translucency Revisited

The Vermin class demonstrates translucency using a package variable, eponymously named `%Vermin`, as its meta-object. If you prefer to use absolutely no package variables beyond those necessary to appease inheritance or possibly the `Exporter`, this strategy is closed to you. That's too bad, because translucent attributes are an appealing technique, so it would be valuable to devise an implementation using only lexicals.

There's a second reason why you might wish to avoid the eponymous package hash. If you use class names with double-colons in them, you would end up poking around somewhere you might not have meant to poke.

```

package Vermin;
$class = "Vermin";
$class->{PopCount}++;
# accesses $Vermin::Vermin{PopCount}

package Vermin::Noxious;

```

```

$class = "Vermin::Noxious";
$class->{PopCount}++;
# accesses $Vermin::Noxious{PopCount}

```

In the first case, because the class name had no double-colons, we got the hash in the current package. But in the second case, instead of getting some hash in the current package, we got the hash %Noxious in the Vermin package. (The noxious vermin just invaded another package and sprayed their data around it. :-) Perl doesn't support relative packages in its naming conventions, so any double-colons trigger a fully-qualified lookup instead of just looking in the current package.

In practice, it is unlikely that the Vermin class had an existing package variable named %Noxious that you just blew away. If you're still mistrustful, you could always stake out your own territory where you know the rules, such as using Eponymous::Vermin::Noxious or Hieronymus::Vermin::Boschious or Leave_Me_Alone::Vermin::Noxious as class names instead. Sure, it's in theory possible that someone else has a class named Eponymous::Vermin with its own %Noxious hash, but this kind of thing is always true. There's no arbiter of package names. It's always the case that globals like @Cwd::ISA would collide if more than one class uses the same Cwd package.

If this still leaves you with an uncomfortable twinge of paranoia, we have another solution for you. There's nothing that says that you have to have a package variable to hold a class meta-object, either for monadic classes or for translucent attributes. Just code up the methods so that they access a lexical instead.

Here's another implementation of the Vermin class with semantics identical to those given previously, but this time using no package variables.

```

package Vermin;

# Here's the class meta-object, eponymously named.
# It holds all class data, and also all instance data
# so the latter can be used for both initialization
# and translucency.  it's a template.
my %ClassData = (
    PopCount => 0,          # capital for class attributes
    color    => "beige",    # small for instance attributes
);

# constructor method
# invoked as class method or object method
sub spawn {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $self = {};
    bless($self, $class);
    $ClassData{PopCount}++;
    # init fields from invoking object, or omit if
    # invoking object is the class to provide translucency
    %$self = %$obclass if ref $obclass;
    return $self;
}

# translucent accessor for "color" attribute
# invoked as class method or object method
sub color {
    my $self = shift;

    # handle class invocation
    unless (ref $self) {
        $ClassData{color} = shift if @_;
        return $ClassData{color}
    }
}

```

```

    }

    # handle object invocation
    $self->{color} = shift if @_;
    if (defined $self->{color}) { # not exists!
        return $self->{color};
    } else {
        return $ClassData{color};
    }
}

# class attribute accessor for "PopCount" attribute
# invoked as class method or object method
sub population {
    return $ClassData{PopCount};
}

# instance destructor; invoked only as object method
sub DESTROY {
    $ClassData{PopCount}--;
}

# detect whether an object attribute is translucent
# (typically?) invoked only as object method
sub is_translucent {
    my($self, $attr) = @_;
    $self = \%ClassData if !ref $self;
    return !defined $self->{$attr};
}

# test for presence of attribute in class
# invoked as class method or object method
sub has_attribute {
    my($self, $attr) = @_;
    return exists $ClassData{$attr};
}

```

NOTES

Inheritance is a powerful but subtle device, best used only after careful forethought and design. Aggregation instead of inheritance is often a better approach.

You can't use file-scoped lexicals in conjunction with the SelfLoader or the AutoLoader, because they alter the lexical scope in which the module's methods wind up getting compiled.

The usual mealy-mouthed package-mungeing doubtless applies to setting up names of object attributes. For example, `< $self-{ObData1}` should probably be `< $self-{ __PACKAGE__ . "_ObData1" }`, but that would just confuse the examples.

SEE ALSO

[perltoot](#), [perlobj](#), [perlmod](#), and [perlbot](#).

The Tie::SecureHash and Class::Data::Inheritable modules from CPAN are worth checking out.

AUTHOR AND COPYRIGHT

Copyright (c) 1999 Tom Christiansen. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A

simple comment in the code giving credit would be courteous but is not required.

ACKNOWLEDGEMENTS

Russ Allbery, Jon Orwant, Randy Ray, Larry Rosler, Nat Torkington, and Stephen Warren all contributed suggestions and corrections to this piece. Thanks especially to Damian Conway for his ideas and feedback, and without whose indirect prodding I might never have taken the time to show others how much Perl has to offer in the way of objects once you start thinking outside the tiny little box that today's "popular" object-oriented languages enforce.

HISTORY

Last edit: Sun Feb 4 20:50:28 EST 2001

NAME

perltoot – Tom's object-oriented tutorial for perl

DESCRIPTION

Object-oriented programming is a big seller these days. Some managers would rather have objects than sliced bread. Why is that? What's so special about an object? Just what *is* an object anyway?

An object is nothing but a way of tucking away complex behaviours into a neat little easy-to-use bundle. (This is what professors call abstraction.) Smart people who have nothing to do but sit around for weeks on end figuring out really hard problems make these nifty objects that even regular people can use. (This is what professors call software reuse.) Users (well, programmers) can play with this little bundle all they want, but they aren't to open it up and mess with the insides. Just like an expensive piece of hardware, the contract says that you void the warranty if you muck with the cover. So don't do that.

The heart of objects is the class, a protected little private namespace full of data and functions. A class is a set of related routines that addresses some problem area. You can think of it as a user-defined type. The Perl package mechanism, also used for more traditional modules, is used for class modules as well. Objects "live" in a class, meaning that they belong to some package.

More often than not, the class provides the user with little bundles. These bundles are objects. They know whose class they belong to, and how to behave. Users ask the class to do something, like "give me an object." Or they can ask one of these objects to do something. Asking a class to do something for you is calling a *class method*. Asking an object to do something for you is calling an *object method*. Asking either a class (usually) or an object (sometimes) to give you back an object is calling a *constructor*, which is just a kind of method.

That's all well and good, but how is an object different from any other Perl data type? Just what is an object *really*; that is, what's its fundamental type? The answer to the first question is easy. An object is different from any other data type in Perl in one and only one way: you may dereference it using not merely string or numeric subscripts as with simple arrays and hashes, but with named subroutine calls. In a word, with *methods*.

The answer to the second question is that it's a reference, and not just any reference, mind you, but one whose referent has been *bless()* ed into a particular class (read: package). What kind of reference? Well, the answer to that one is a bit less concrete. That's because in Perl the designer of the class can employ any sort of reference they'd like as the underlying intrinsic data type. It could be a scalar, an array, or a hash reference. It could even be a code reference. But because of its inherent flexibility, an object is usually a hash reference.

Creating a Class

Before you create a class, you need to decide what to name it. That's because the class (package) name governs the name of the file used to house it, just as with regular modules. Then, that class (package) should provide one or more ways to generate objects. Finally, it should provide mechanisms to allow users of its objects to indirectly manipulate these objects from a distance.

For example, let's make a simple Person class module. It gets stored in the file Person.pm. If it were called a Happy::Person class, it would be stored in the file Happy/Person.pm, and its package would become Happy::Person instead of just Person. (On a personal computer not running Unix or Plan 9, but something like Mac OS or VMS, the directory separator may be different, but the principle is the same.) Do not assume any formal relationship between modules based on their directory names. This is merely a grouping convenience, and has no effect on inheritance, variable accessibility, or anything else.

For this module we aren't going to use Exporter, because we're a well-behaved class module that doesn't export anything at all. In order to manufacture objects, a class needs to have a *constructor method*. A constructor gives you back not just a regular data type, but a brand-new object in that class. This magic is taken care of by the *bless()* function, whose sole purpose is to enable its referent to be used as an object. Remember: being an object really means nothing more than that methods may now be called against it.

While a constructor may be named anything you'd like, most Perl programmers seem to like to call theirs `new()`. However, `new()` is not a reserved word, and a class is under no obligation to supply such. Some programmers have also been known to use a function with the same name as the class as the constructor.

Object Representation

By far the most common mechanism used in Perl to represent a Pascal record, a C struct, or a C++ class is an anonymous hash. That's because a hash has an arbitrary number of data fields, each conveniently accessed by an arbitrary name of your own devising.

If you were just doing a simple struct-like emulation, you would likely go about it something like this:

```
$rec = {
    name  => "Jason",
    age   => 23,
    peers => [ "Norbert", "Rhys", "Phineas"],
};
```

If you felt like it, you could add a bit of visual distinction by up-casing the hash keys:

```
$rec = {
    NAME  => "Jason",
    AGE   => 23,
    PEERS => [ "Norbert", "Rhys", "Phineas"],
};
```

And so you could get at `< $rec-{NAME}` to find "Jason", or `< @{ $rec-{PEERS} }` to get at "Norbert", "Rhys", and "Phineas". (Have you ever noticed how many 23-year-old programmers seem to be named "Jason" these days? :-)

This same model is often used for classes, although it is not considered the pinnacle of programming propriety for folks from outside the class to come waltzing into an object, brazenly accessing its data members directly. Generally speaking, an object should be considered an opaque cookie that you use *object methods* to access. Visually, methods look like you're dereffing a reference using a function name instead of brackets or braces.

Class Interface

Some languages provide a formal syntactic interface to a class's methods, but Perl does not. It relies on you to read the documentation of each class. If you try to call an undefined method on an object, Perl won't complain, but the program will trigger an exception while it's running. Likewise, if you call a method expecting a prime number as its argument with a non-prime one instead, you can't expect the compiler to catch this. (Well, you can expect it all you like, but it's not going to happen.)

Let's suppose you have a well-educated user of your `Person` class, someone who has read the docs that explain the prescribed interface. Here's how they might use the `Person` class:

```
use Person;

$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( "Norbert", "Rhys", "Phineas" );

push @All_Recs, $him; # save object in array for later

printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", $him->peers), "\n";

printf "Last rec's name is %s\n", $All_Recs[-1]->name;
```

As you can see, the user of the class doesn't know (or at least, has no business paying attention to the fact) that the object has one particular implementation or another. The interface to the class and its objects is exclusively via methods, and that's all the user of the class should ever play with.

Constructors and Instance Methods

Still, *someone* has to know what's in the object. And that someone is the class. It implements methods that the programmer uses to access the object. Here's how to implement the `Person` class using the standard hash-ref-as-an-object idiom. We'll make a class method called `new()` to act as the constructor, and three object methods called `name()`, `age()`, and `peers()` to get at per-object data hidden away in our anonymous hash.

```
package Person;
use strict;

#####
## the object constructor (simplistic version) ##
#####
sub new {
    my $self = {};
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    bless($self);          # but see below
    return $self;
}

#####
## methods to access per-object data          ##
##                                           ##
## With args, they set the value.  Without ##
## any, they only retrieve it/them.         ##
#####

sub name {
    my $self = shift;
    if (@_) { $self->{NAME} = shift }
    return $self->{NAME};
}

sub age {
    my $self = shift;
    if (@_) { $self->{AGE} = shift }
    return $self->{AGE};
}

sub peers {
    my $self = shift;
    if (@_) { @{ $self->{PEERS} } = @_ }
    return @{ $self->{PEERS} };
}

1; # so the require or use succeeds
```

We've created three methods to access an object's data, `name()`, `age()`, and `peers()`. These are all substantially similar. If called with an argument, they set the appropriate field; otherwise they return the value held by that field, meaning the value of that hash key.

Planning for the Future: Better Constructors

Even though at this point you may not even know what it means, someday you're going to worry about inheritance. (You can safely ignore this for now and worry about it later if you'd like.) To ensure that this all works out smoothly, you must use the double-argument form of `bless()`. The second argument is the class into which the referent will be blessed. By not assuming our own class as the default second argument

and instead using the class passed into us, we make our constructor inheritable.

While we're at it, let's make our constructor a bit more flexible. Rather than being uniquely a class method, we'll set it up so that it can be called as either a class method *or* an object method. That way you can say:

```
$me = Person->new();
$him = $me->new();
```

To do this, all we have to do is check whether what was passed in was a reference or not. If so, we were invoked as an object method, and we need to extract the package (class) using the `ref()` function. If not, we just use the string passed in as the package name for blessing our referent.

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {};
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    bless ($self, $class);
    return $self;
}
```

That's about all there is for constructors. These methods bring objects to life, returning neat little opaque bundles to the user to be used in subsequent method calls.

Destructors

Every story has a beginning and an end. The beginning of the object's story is its constructor, explicitly called when the object comes into existence. But the ending of its story is the *destructor*, a method implicitly called when an object leaves this life. Any per-object clean-up code is placed in the destructor, which must (in Perl) be called DESTROY.

If constructors can have arbitrary names, then why not destructors? Because while a constructor is explicitly called, a destructor is not. Destruction happens automatically via Perl's garbage collection (GC) system, which is a quick but somewhat lazy reference-based GC system. To know what to call, Perl insists that the destructor be named DESTROY. Perl's notion of the right time to call a destructor is not well-defined currently, which is why your destructors should not rely on when they are called.

Why is DESTROY in all caps? Perl on occasion uses purely uppercase function names as a convention to indicate that the function will be automatically called by Perl in some way. Others that are called implicitly include BEGIN, END, AUTOLOAD, plus all methods used by tied objects, described in [perltie](#).

In really good object-oriented programming languages, the user doesn't care when the destructor is called. It just happens when it's supposed to. In low-level languages without any GC at all, there's no way to depend on this happening at the right time, so the programmer must explicitly call the destructor to clean up memory and state, crossing their fingers that it's the right time to do so. Unlike C++, an object destructor is nearly never needed in Perl, and even when it is, explicit invocation is uncalled for. In the case of our Person class, we don't need a destructor because Perl takes care of simple matters like memory deallocation.

The only situation where Perl's reference-based GC won't work is when there's a circularity in the data structure, such as:

```
$this->{WHATEVER} = $this;
```

In that case, you must delete the self-reference manually if you expect your program not to leak memory. While admittedly error-prone, this is the best we can do right now. Nonetheless, rest assured that when your program is finished, its objects' destructors are all duly called. So you are guaranteed that an object *eventually* gets properly destroyed, except in the unique case of a program that never exits. (If you're running Perl embedded in another application, this full GC pass happens a bit more frequently—whenever a thread shuts down.)

Other Object Methods

The methods we've talked about so far have either been constructors or else simple "data methods", interfaces to data stored in the object. These are a bit like an object's data members in the C++ world, except that strangers don't access them as data. Instead, they should only access the object's data indirectly via its methods. This is an important rule: in Perl, access to an object's data should *only* be made through methods.

Perl doesn't impose restrictions on who gets to use which methods. The public-versus-private distinction is by convention, not syntax. (Well, unless you use the `Alias` module described below in

[Data Members as Variables](#).) Occasionally you'll see method names beginning or ending with an underscore or two. This marking is a convention indicating that the methods are private to that class alone and sometimes to its closest acquaintances, its immediate subclasses. But this distinction is not enforced by Perl itself. It's up to the programmer to behave.

There's no reason to limit methods to those that simply access data. Methods can do anything at all. The key point is that they're invoked against an object or a class. Let's say we'd like object methods that do more than fetch or set one particular field.

```
sub exclaim {
    my $self = shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $self->{NAME}, $self->{AGE}, join(", ", @{$self->{PEERS}});
}
```

Or maybe even one like this:

```
sub happy_birthday {
    my $self = shift;
    return ++$self->{AGE};
}
```

Some might argue that one should go at these this way:

```
sub exclaim {
    my $self = shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $self->name, $self->age, join(", ", $self->peers);
}

sub happy_birthday {
    my $self = shift;
    return $self->age( $self->age() + 1 );
}
```

But since these methods are all executing in the class itself, this may not be critical. There are tradeoffs to be made. Using direct hash access is faster (about an order of magnitude faster, in fact), and it's more convenient when you want to interpolate in strings. But using methods (the external interface) internally shields not just the users of your class but even you yourself from changes in your data representation.

Class Data

What about "class data", data items common to each object in a class? What would you want that for? Well, in your `Person` class, you might like to keep track of the total people alive. How do you implement that?

You *could* make it a global variable called `$Person::Census`. But about only reason you'd do that would be if you *wanted* people to be able to get at your class data directly. They could just say `$Person::Census` and play around with it. Maybe this is ok in your design scheme. You might even conceivably want to make it an exported variable. To be exportable, a variable must be a (package) global. If this were a traditional module rather than an object-oriented one, you might do that.

While this approach is expected in most traditional modules, it's generally considered rather poor form in most object modules. In an object module, you should set up a protective veil to separate interface from

implementation. So provide a class method to access class data just as you provide object methods to access object data.

So, you *could* still keep `$Census` as a package global and rely upon others to honor the contract of the module and therefore not play around with its implementation. You could even be supertricky and make `$Census` a tied object as described in [perlty](#), thereby intercepting all accesses.

But more often than not, you just want to make your class data a file-scoped lexical. To do so, simply put this at the top of the file:

```
my $Census = 0;
```

Even though the scope of a `my()` normally expires when the block in which it was declared is done (in this case the whole file being required or used), Perl's deep binding of lexical variables guarantees that the variable will not be deallocated, remaining accessible to functions declared within that scope. This doesn't work with global variables given temporary values via `local()`, though.

Irrespective of whether you leave `$Census` a package global or make it instead a file-scoped lexical, you should make these changes to your `Person::new()` constructor:

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {};
    $Census++;
    $self->{NAME} = undef;
    $self->{AGE} = undef;
    $self->{PEERS} = [];
    bless ($self, $class);
    return $self;
}

sub population {
    return $Census;
}
```

Now that we've done this, we certainly do need a destructor so that when `Person` is destroyed, the `$Census` goes down. Here's how this could be done:

```
sub DESTROY { --$Census }
```

Notice how there's no memory to deallocate in the destructor? That's something that Perl takes care of for you all by itself.

Alternatively, you could use the `Class::Data::Inheritable` module from CPAN.

Accessing Class Data

It turns out that this is not really a good way to go about handling class data. A good scalable rule is that *you must never reference class data directly from an object method*. Otherwise you aren't building a scalable, inheritable class. The object must be the rendezvous point for all operations, especially from an object method. The globals (class data) would in some sense be in the "wrong" package in your derived classes. In Perl, methods execute in the context of the class they were defined in, *not* that of the object that triggered them. Therefore, namespace visibility of package globals in methods is unrelated to inheritance.

Got that? Maybe not. Ok, let's say that some other class "borrowed" (well, inherited) the `DESTROY` method as it was defined above. When those objects are destroyed, the original `$Census` variable will be altered, not the one in the new class's package namespace. Perhaps this is what you want, but probably it isn't.

Here's how to fix this. We'll store a reference to the data in the value accessed by the hash key `"_CENSUS"`. Why the underscore? Well, mostly because an initial underscore already conveys strong

feelings of magicalness to a C programmer. It's really just a mnemonic device to remind ourselves that this field is special and not to be used as a public data member in the same way that NAME, AGE, and PEERS are. (Because we've been developing this code under the strict pragma, prior to perl version 5.004 we'll have to quote the field name.)

```
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {};
    $self->{NAME}      = undef;
    $self->{AGE}       = undef;
    $self->{PEERS}     = [];
    # "private" data
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub population {
    my $self = shift;
    if (ref $self) {
        return ${ $self->{"_CENSUS"} };
    } else {
        return $Census;
    }
}

sub DESTROY {
    my $self = shift;
    -- ${ $self->{"_CENSUS"} };
}
```

Debugging Methods

It's common for a class to have a debugging mechanism. For example, you might want to see when objects are created or destroyed. To do that, add a debugging variable as a file-scoped lexical. For this, we'll pull in the standard Carp module to emit our warnings and fatal messages. That way messages will come out with the caller's filename and line number instead of our own; if we wanted them to be from our own perspective, we'd just use `die()` and `warn()` directly instead of `croak()` and `carp()` respectively.

```
use Carp;
my $Debugging = 0;
```

Now add a new class method to access the variable.

```
sub debug {
    my $class = shift;
    if (ref $class) { confess "Class method called as object method" }
    unless (@_ == 1) { confess "usage: CLASSNAME->debug(level)" }
    $Debugging = shift;
}
```

Now fix up DESTROY to murmur a bit as the moribund object expires:

```
sub DESTROY {
    my $self = shift;
    if ($Debugging) { carp "Destroying $self " . $self->name }
    -- ${ $self->{"_CENSUS"} };
}
```

One could conceivably make a per-object debug state. That way you could call both of these:

```
Person->debug(1);    # entire class
$him->debug(1);      # just this object
```

To do so, we need our debugging method to be a "bimodal" one, one that works on both classes *and* objects. Therefore, adjust the debug() and DESTROY methods as follows:

```
sub debug {
    my $self = shift;
    confess "usage: thing->debug(level)" unless @_ == 1;
    my $level = shift;
    if (ref($self)) {
        $self->{"_DEBUG"} = $level;      # just myself
    } else {
        $Debugging        = $level;      # whole class
    }
}

sub DESTROY {
    my $self = shift;
    if ($Debugging || $self->{"_DEBUG"}) {
        carp "Destroying $self " . $self->name;
    }
    -- ${ $self->{"_CENSUS"} };
}
```

What happens if a derived class (which we'll call Employee) inherits methods from this Person base class? Then `< Employee->debug()`, when called as a class method, manipulates `$Person::Debugging` not `$Employee::Debugging`.

Class Destructors

The object destructor handles the death of each distinct object. But sometimes you want a bit of cleanup when the entire class is shut down, which currently only happens when the program exits. To make such a *class destructor*, create a function in that class's package named END. This works just like the END function in traditional modules, meaning that it gets called whenever your program exits unless it execs or dies of an uncaught signal. For example,

```
sub END {
    if ($Debugging) {
        print "All persons are going away now.\n";
    }
}
```

When the program exits, all the class destructors (END functions) are be called in the opposite order that they were loaded in (LIFO order).

Documenting the Interface

And there you have it: we've just shown you the *implementation* of this Person class. Its *interface* would be its documentation. Usually this means putting it in pod ("plain old documentation") format right there in the same file. In our Person example, we would place the following docs anywhere in the Person.pm file. Even though it looks mostly like code, it's not. It's embedded documentation such as would be used by the pod2man, pod2html, or pod2text programs. The Perl compiler ignores pods entirely, just as the translators ignore code. Here's an example of some pods describing the informal interface:

```
=head1 NAME

Person - class to implement people

=head1 SYNOPSIS
```

```

use Person;

#####
# class methods #
#####
$obj = Person->new;
$count = Person->population;

#####
# object data methods #
#####

### get versions ###
$who = $obj->name;
$years = $obj->age;
@pals = $obj->peers;

### set versions ###
$obj->name("Jason");
$obj->age(23);
$obj->peers("Norbert", "Rhys", "Phineas");

#####
# other object methods #
#####

$phrase = $obj->exclaim;
$obj->happy_birthday;

=head1 DESCRIPTION

```

The Person class implements dah dee dah dee dah....

That's all there is to the matter of interface versus implementation. A programmer who opens up the module and plays around with all the private little shiny bits that were safely locked up behind the interface contract has voided the warranty, and you shouldn't worry about their fate.

Aggregation

Suppose you later want to change the class to implement better names. Perhaps you'd like to support both given names (called Christian names, irrespective of one's religion) and family names (called surnames), plus nicknames and titles. If users of your Person class have been properly accessing it through its documented interface, then you can easily change the underlying implementation. If they haven't, then they lose and it's their fault for breaking the contract and voiding their warranty.

To do this, we'll make another class, this one called Fullname. What's the Fullname class look like? To answer that question, you have to first figure out how you want to use it. How about we use it this way:

```

$him = Person->new();
$him->fullname->title("St");
$him->fullname->christian("Thomas");
$him->fullname->surname("Aquinas");
$him->fullname->nickname("Tommy");
printf "His normal name is %s\n", $him->name;
printf "But his real name is %s\n", $him->fullname->as_string;

```

Ok. To do this, we'll change Person::new() so that it supports a full name field this way:

```

sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {};

```



```

    $self->{FULLNAME} = Fullname->new();
    $self->{AGE}       = undef;
    $self->{PEERS}     = [];
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub fullname {
    my $self = shift;
    return $self->{FULLNAME};
}

```

Then to support old code, define `Person::name()` this way:

```

sub name {
    my $self = shift;
    return $self->{FULLNAME}->nickname(@_)
        || $self->{FULLNAME}->christian(@_);
}

```

Here's the `Fullname` class. We'll use the same technique of using a hash reference to hold data fields, and methods by the appropriate name to access them:

```

package Fullname;
use strict;

sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {
        TITLE      => undef,
        CHRISTIAN  => undef,
        SURNAME    => undef,
        NICK       => undef,
    };
    bless ($self, $class);
    return $self;
}

sub christian {
    my $self = shift;
    if (@_) { $self->{CHRISTIAN} = shift }
    return $self->{CHRISTIAN};
}

sub surname {
    my $self = shift;
    if (@_) { $self->{SURNAME} = shift }
    return $self->{SURNAME};
}

sub nickname {
    my $self = shift;
    if (@_) { $self->{NICK} = shift }
    return $self->{NICK};
}

```

```

sub title {
    my $self = shift;
    if (@_) { $self->{TITLE} = shift }
    return $self->{TITLE};
}

sub as_string {
    my $self = shift;
    my $name = join(" ", @$self{'CHRISTIAN', 'SURNAME'});
    if ($self->{TITLE}) {
        $name = $self->{TITLE} . " " . $name;
    }
    return $name;
}

1;

```

Finally, here's the test program:

```

#!/usr/bin/perl -w
use strict;
use Person;
sub END { show_census() }

sub show_census () {
    printf "Current population: %d\n", Person->population;
}

Person->debug(1);

show_census();

my $him = Person->new();

$him->fullname->christian("Thomas");
$him->fullname->surname("Aquinas");
$him->fullname->nickname("Tommy");
$him->fullname->title("St");
$him->age(1);

printf "%s is really %s.\n", $him->name, $him->fullname;
printf "%s's age: %d.\n", $him->name, $him->age;
$him->happy_birthday;
printf "%s's age: %d.\n", $him->name, $him->age;

show_census();

```

Inheritance

Object-oriented programming systems all support some notion of inheritance. Inheritance means allowing one class to piggy-back on top of another one so you don't have to write the same code again and again. It's about software reuse, and therefore related to Laziness, the principal virtue of a programmer. (The import/export mechanisms in traditional modules are also a form of code reuse, but a simpler one than the true inheritance that you find in object modules.)

Sometimes the syntax of inheritance is built into the core of the language, and sometimes it's not. Perl has no special syntax for specifying the class (or classes) to inherit from. Instead, it's all strictly in the semantics. Each package can have a variable called `@ISA`, which governs (method) inheritance. If you try to call a method on an object or class, and that method is not found in that object's package, Perl then looks to `@ISA` for other packages to go looking through in search of the missing method.

Like the special per-package variables recognized by `Exporter` (such as `@EXPORT`, `@EXPORT_OK`,

`@EXPORT_FAIL`, `%EXPORT_TAGS`, and `$VERSION`), the `@ISA` array *must* be a package-scoped global and not a file-scoped lexical created via `my()`. Most classes have just one item in their `@ISA` array. In this case, we have what's called "single inheritance", or SI for short.

Consider this class:

```
package Employee;
use Person;
@ISA = ("Person");
1;
```

Not a lot to it, eh? All it's doing so far is loading in another class and stating that this one will inherit methods from that other class if need be. We have given it none of its own methods. We rely upon an `Employee` to behave just like a `Person`.

Setting up an empty class like this is called the "empty subclass test"; that is, making a derived class that does nothing but inherit from a base class. If the original base class has been designed properly, then the new derived class can be used as a drop-in replacement for the old one. This means you should be able to write a program like this:

```
use Employee;
my $empl = Employee->new();
$empl->name("Jason");
$empl->age(23);
printf "%s is age %d.\n", $empl->name, $empl->age;
```

By proper design, we mean always using the two-argument form of `bless()`, avoiding direct access of global data, and not exporting anything. If you look back at the `Person::new()` function we defined above, we were careful to do that. There's a bit of package data used in the constructor, but the reference to this is stored on the object itself and all other methods access package data via that reference, so we should be ok.

What do we mean by the `Person::new()` function — isn't that actually a method? Well, in principle, yes. A method is just a function that expects as its first argument a class name (package) or object (blessed reference). `Person::new()` is the function that both the `< Person-new()` method and the `< Employee-new()` method end up calling. Understand that while a method call looks a lot like a function call, they aren't really quite the same, and if you treat them as the same, you'll very soon be left with nothing but broken programs. First, the actual underlying calling conventions are different: method calls get an extra argument. Second, function calls don't do inheritance, but methods do.

Method Call	Resulting Function Call
-----	-----
<code>Person->new()</code>	<code>Person::new("Person")</code>
<code>Employee->new()</code>	<code>Person::new("Employee")</code>

So don't use function calls when you mean to call a method.

If an employee is just a `Person`, that's not all too very interesting. So let's add some other methods. We'll give our employee data fields to access their salary, their employee ID, and their start date.

If you're getting a little tired of creating all these nearly identical methods just to get at the object's data, do not despair. Later, we'll describe several different convenience mechanisms for shortening this up. Meanwhile, here's the straight-forward way:

```
sub salary {
    my $self = shift;
    if (@_) { $self->{SALARY} = shift }
    return $self->{SALARY};
}
```

```

sub id_number {
    my $self = shift;
    if (@_) { $self->{ID} = shift }
    return $self->{ID};
}

sub start_date {
    my $self = shift;
    if (@_) { $self->{START_DATE} = shift }
    return $self->{START_DATE};
}

```

Overridden Methods

What happens when both a derived class and its base class have the same method defined? Well, then you get the derived class's version of that method. For example, let's say that we want the `peers()` method called on an employee to act a bit differently. Instead of just returning the list of peer names, let's return slightly different strings. So doing this:

```

$empl->peers("Peter", "Paul", "Mary");
printf "His peers are: %s\n", join(", ", $empl->peers);

```

will produce:

```

His peers are: PEON=PETER, PEON=PAUL, PEON=MARY

```

To do this, merely add this definition into the `Employee.pm` file:

```

sub peers {
    my $self = shift;
    if (@_) { @{ $self->{PEERS} } = @_ }
    return map { "PEON=\U$_" } @{ $self->{PEERS} };
}

```

There, we've just demonstrated the high-falutin' concept known in certain circles as *polymorphism*. We've taken on the form and behaviour of an existing object, and then we've altered it to suit our own purposes. This is a form of Laziness. (Getting polymorphed is also what happens when the wizard decides you'd look better as a frog.)

Every now and then you'll want to have a method call trigger both its derived class (also known as "subclass") version as well as its base class (also known as "superclass") version. In practice, constructors and destructors are likely to want to do this, and it probably also makes sense in the `debug()` method we showed previously.

To do this, add this to `Employee.pm`:

```

use Carp;
my $Debugging = 0;

sub debug {
    my $self = shift;
    confess "usage: thing->debug(level)" unless @_ == 1;
    my $level = shift;
    if (ref($self)) {
        $self->{"_DEBUG"} = $level;
    } else {
        $Debugging = $level;          # whole class
    }
    Person::debug($self, $Debugging); # don't really do this
}

```

As you see, we turn around and call the `Person` package's `debug()` function. But this is far too fragile for good design. What if `Person` doesn't have a `debug()` function, but is inheriting *its* `debug()` method from elsewhere? It would have been slightly better to say

```
Person->debug($Debugging);
```

But even that's got too much hard-coded. It's somewhat better to say

```
$self->Person::debug($Debugging);
```

Which is a funny way to say to start looking for a `debug()` method up in `Person`. This strategy is more often seen on overridden object methods than on overridden class methods.

There is still something a bit off here. We've hard-coded our superclass's name. This in particular is bad if you change which classes you inherit from, or add others. Fortunately, the pseudoclass `SUPER` comes to the rescue here.

```
$self->SUPER::debug($Debugging);
```

This way it starts looking in my class's `@ISA`. This only makes sense from *within* a method call, though. Don't try to access anything in `SUPER::` from anywhere else, because it doesn't exist outside an overridden method call.

Things are getting a bit complicated here. Have we done anything we shouldn't? As before, one way to test whether we're designing a decent class is via the empty subclass test. Since we already have an `Employee` class that we're trying to check, we'd better get a new empty subclass that can derive from `Employee`. Here's one:

```
package Boss;
use Employee;          # :-)
@ISA = qw(Employee);
```

And here's the test program:

```
#!/usr/bin/perl -w
use strict;
use Boss;
Boss->debug(1);

my $boss = Boss->new();

$boss->fullname->title("Don");
$boss->fullname->surname("Pichon Alvarez");
$boss->fullname->christian("Federico Jesus");
$boss->fullname->nickname("Fred");

$boss->age(47);
$boss->peers("Frank", "Felipe", "Faust");

printf "%s is age %d.\n", $boss->fullname, $boss->age;
printf "His peers are: %s\n", join(", ", $boss->peers);
```

Running it, we see that we're still ok. If you'd like to dump out your object in a nice format, somewhat like the way the `'x'` command works in the debugger, you could use the `Data::Dumper` module from CPAN this way:

```
use Data::Dumper;
print "Here's the boss:\n";
print Dumper($boss);
```

Which shows us something like this:

```
Here's the boss:
$VAR1 = bless( {
```

```

_CENSUS => \1,
FULLNAME => bless( {
    TITLE => 'Don',
    SURNAME => 'Pichon Alvarez',
    NICK => 'Fred',
    CHRISTIAN => 'Federico Jesus'
}, 'Fullname' ),
AGE => 47,
PEERS => [
    'Frank',
    'Felipe',
    'Faust'
]
}, 'Boss' );

```

Hm.... something's missing there. What about the salary, start date, and ID fields? Well, we never set them to anything, even undef, so they don't show up in the hash's keys. The Employee class has no new() method of its own, and the new() method in Person doesn't know about Employees. (Nor should it: proper OO design dictates that a subclass be allowed to know about its immediate superclass, but never vice-versa.) So let's fix up Employee::new() this way:

```

sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new();
    $self->{SALARY} = undef;
    $self->{ID} = undef;
    $self->{START_DATE} = undef;
    bless ($self, $class);      # reconsecrate
    return $self;
}

```

Now if you dump out an Employee or Boss object, you'll find that new fields show up there now.

Multiple Inheritance

Ok, at the risk of confusing beginners and annoying OO gurus, it's time to confess that Perl's object system includes that controversial notion known as multiple inheritance, or MI for short. All this means is that rather than having just one parent class who in turn might itself have a parent class, etc., that you can directly inherit from two or more parents. It's true that some uses of MI can get you into trouble, although hopefully not quite so much trouble with Perl as with dubiously-OO languages like C++.

The way it works is actually pretty simple: just put more than one package name in your @ISA array. When it comes time for Perl to go finding methods for your object, it looks at each of these packages in order. Well, kinda. It's actually a fully recursive, depth-first order. Consider a bunch of @ISA arrays like this:

```

@First::ISA = qw( Alpha );
@Second::ISA = qw( Beta );
@Third::ISA = qw( First Second );

```

If you have an object of class Third:

```

my $ob = Third->new();
$ob->spin();

```

How do we find a spin() method (or a new() method for that matter)? Because the search is depth-first, classes will be looked up in the following order: Third, First, Alpha, Second, and Beta.

In practice, few class modules have been seen that actually make use of MI. One nearly always chooses simple containership of one class within another over MI. That's why our Person object *contained* a

Fullname object. That doesn't mean it *was* one.

However, there is one particular area where MI in Perl is rampant: borrowing another class's class methods. This is rather common, especially with some bundled "objectless" classes, like `Exporter`, `DynaLoader`, `AutoLoader`, and `SelfLoader`. These classes do not provide constructors; they exist only so you may inherit their class methods. (It's not entirely clear why inheritance was done here rather than traditional module importation.)

For example, here is the `POSIX` module's `@ISA`:

```
package POSIX;
@ISA = qw(Exporter DynaLoader);
```

The `POSIX` module isn't really an object module, but then, neither are `Exporter` or `DynaLoader`. They're just lending their classes' behaviours to `POSIX`.

Why don't people use MI for object methods much? One reason is that it can have complicated side-effects. For one thing, your inheritance graph (no longer a tree) might converge back to the same base class. Although Perl guards against recursive inheritance, merely having parents who are related to each other via a common ancestor, incestuous though it sounds, is not forbidden. What if in our Third class shown above we wanted its `new()` method to also call both overridden constructors in its two parent classes? The `SUPER` notation would only find the first one. Also, what about if the `Alpha` and `Beta` classes both had a common ancestor, like `Nought`? If you kept climbing up the inheritance tree calling overridden methods, you'd end up calling `Nought::new()` twice, which might well be a bad idea.

UNIVERSAL: The Root of All Objects

Wouldn't it be convenient if all objects were rooted at some ultimate base class? That way you could give every object common methods without having to go and add it to each and every `@ISA`. Well, it turns out that you can. You don't see it, but Perl tacitly and irrevocably assumes that there's an extra element at the end of `@ISA`: the class `UNIVERSAL`. In version 5.003, there were no predefined methods there, but you could put whatever you felt like into it.

However, as of version 5.004 (or some subversive releases, like 5.003_08), `UNIVERSAL` has some methods in it already. These are builtin to your Perl binary, so they don't take any extra time to load. Predefined methods include `isa()`, `can()`, and `VERSION()`. `isa()` tells you whether an object or class "is" another one without having to traverse the hierarchy yourself:

```
$has_io = $fd->isa("IO::Handle");
$itza_handle = IO::Socket->isa("IO::Handle");
```

The `can()` method, called against that object or class, reports back whether its string argument is a callable method name in that class. In fact, it gives you back a function reference to that method:

```
$this_print_method = $obj->can('as_string');
```

Finally, the `VERSION` method checks whether the class (or the object's class) has a package global called `$VERSION` that's high enough, as in:

```
Some_Module->VERSION(3.0);
$this_vers = $ob->VERSION();
```

However, we don't usually call `VERSION` ourselves. (Remember that an all uppercase function name is a Perl convention that indicates that the function will be automatically used by Perl in some way.) In this case, it happens when you say

```
use Some_Module 3.0;
```

If you wanted to add version checking to your `Person` class explained above, just add this to `Person.pm`:

```
our $VERSION = '1.1';
```

and then in `Employee.pm` could you can say

```
use Employee 1.1;
```

And it would make sure that you have at least that version number or higher available. This is not the same as loading in that exact version number. No mechanism currently exists for concurrent installation of multiple versions of a module. Lamentably.

Alternate Object Representations

Nothing requires objects to be implemented as hash references. An object can be any sort of reference so long as its referent has been suitably blessed. That means scalar, array, and code references are also fair game.

A scalar would work if the object has only one datum to hold. An array would work for most cases, but makes inheritance a bit dodgy because you have to invent new indices for the derived classes.

Arrays as Objects

If the user of your class honors the contract and sticks to the advertised interface, then you can change its underlying interface if you feel like it. Here's another implementation that conforms to the same interface specification. This time we'll use an array reference instead of a hash reference to represent the object.

```
package Person;
use strict;

my($NAME, $AGE, $PEERS) = ( 0 .. 2 );

#####
## the object constructor (array version) ##
#####
sub new {
    my $self = [];
    $self->[$NAME] = undef; # this is unnecessary
    $self->[$AGE] = undef; # as is this
    $self->[$PEERS] = []; # but this isn't, really
    bless($self);
    return $self;
}

sub name {
    my $self = shift;
    if (@_) { $self->[$NAME] = shift }
    return $self->[$NAME];
}

sub age {
    my $self = shift;
    if (@_) { $self->[$AGE] = shift }
    return $self->[$AGE];
}

sub peers {
    my $self = shift;
    if (@_) { @{ $self->[$PEERS] } = @_ }
    return @{ $self->[$PEERS] };
}

1; # so the require or use succeeds
```

You might guess that the array access would be a lot faster than the hash access, but they're actually comparable. The array is a *little* bit faster, but not more than ten or fifteen percent, even when you replace the variables above like \$AGE with literal numbers, like 1. A bigger difference between the two approaches can be found in memory use. A hash representation takes up more memory than an array representation

because you have to allocate memory for the keys as well as for the values. However, it really isn't that bad, especially since as of version 5.004, memory is only allocated once for a given hash key, no matter how many hashes have that key. It's expected that sometime in the future, even these differences will fade into obscurity as more efficient underlying representations are devised.

Still, the tiny edge in speed (and somewhat larger one in memory) is enough to make some programmers choose an array representation for simple classes. There's still a little problem with scalability, though, because later in life when you feel like creating subclasses, you'll find that hashes just work out better.

Closures as Objects

Using a code reference to represent an object offers some fascinating possibilities. We can create a new anonymous function (closure) who alone in all the world can see the object's data. This is because we put the data into an anonymous hash that's lexically visible only to the closure we create, bless, and return as the object. This object's methods turn around and call the closure as a regular subroutine call, passing it the field we want to affect. (Yes, the double-function call is slow, but if you wanted fast, you wouldn't be using objects at all, eh? :-)

Use would be similar to before:

```
use Person;
$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( [ "Norbert", "Rhys", "Phineas" ] );
printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(" ", @{$him->peers}), "\n";
```

but the implementation would be radically, perhaps even sublimely different:

```
package Person;
sub new {
    my $that = shift;
    my $class = ref($that) || $that;
    my $self = {
        NAME => undef,
        AGE  => undef,
        PEERS => [],
    };
    my $closure = sub {
        my $field = shift;
        if (@_) { $self->{$field} = shift }
        return $self->{$field};
    };
    bless($closure, $class);
    return $closure;
}

sub name { &{ $_[0] }("NAME", @_[ 1 .. $#_ ] ) }
sub age { &{ $_[0] }("AGE", @_[ 1 .. $#_ ] ) }
sub peers { &{ $_[0] }("PEERS", @_[ 1 .. $#_ ] ) }

1;
```

Because this object is hidden behind a code reference, it's probably a bit mysterious to those whose background is more firmly rooted in standard procedural or object-based programming languages than in functional programming languages whence closures derive. The object created and returned by the `new()` method is itself not a data reference as we've seen before. It's an anonymous code reference that has within it access to a specific version (lexical binding and instantiation) of the object's data, which are stored in the

private variable `$self`. Although this is the same function each time, it contains a different version of `$self`.

When a method like `$him->name("Jason")` is called, its implicit zeroth argument is the invoking object—just as it is with all method calls. But in this case, it's our code reference (something like a function pointer in C++, but with deep binding of lexical variables). There's not a lot to be done with a code reference beyond calling it, so that's just what we do when we say `&{$_[0]}`. This is just a regular function call, not a method call. The initial argument is the string "NAME", and any remaining arguments are whatever had been passed to the method itself.

Once we're executing inside the closure that had been created in `new()`, the `$self` hash reference suddenly becomes visible. The closure grabs its first argument ("NAME" in this case because that's what the `name()` method passed it), and uses that string to subscript into the private hash hidden in its unique version of `$self`.

Nothing under the sun will allow anyone outside the executing method to be able to get at this hidden data. Well, nearly nothing. You *could* single step through the program using the debugger and find out the pieces while you're in the method, but everyone else is out of luck.

There, if that doesn't excite the Scheme folks, then I just don't know what will. Translation of this technique into C++, Java, or any other braindead-static language is left as a futile exercise for aficionados of those camps.

You could even add a bit of nosiness via the `caller()` function and make the closure refuse to operate unless called via its own package. This would no doubt satisfy certain fastidious concerns of programming police and related puritans.

If you were wondering when Hubris, the third principle virtue of a programmer, would come into play, here you have it. (More seriously, Hubris is just the pride in craftsmanship that comes from having written a sound bit of well-designed code.)

AUTOLOAD: Proxy Methods

Autoloading is a way to intercept calls to undefined methods. An autoload routine may choose to create a new function on the fly, either loaded from disk or perhaps just `eval()`ed right there. This define-on-the-fly strategy is why it's called autoloading.

But that's only one possible approach. Another one is to just have the autoloader method itself directly provide the requested service. When used in this way, you may think of autoloader methods as "proxy" methods.

When Perl tries to call an undefined function in a particular package and that function is not defined, it looks for a function in that same package called `AUTOLOAD`. If one exists, it's called with the same arguments as the original function would have had. The fully-qualified name of the function is stored in that package's global variable `$AUTOLOAD`. Once called, the function can do anything it would like, including defining a new function by the right name, and then doing a really fancy kind of `goto` right to it, erasing itself from the call stack.

What does this have to do with objects? After all, we keep talking about functions, not methods. Well, since a method is just a function with an extra argument and some fancier semantics about where it's found, we can use autoloading for methods, too. Perl doesn't start looking for an `AUTOLOAD` method until it has exhausted the recursive hunt up through `@ISA`, though. Some programmers have even been known to define a `UNIVERSAL::AUTOLOAD` method to trap unresolved method calls to any kind of object.

Autoloaded Data Methods

You probably began to get a little suspicious about the duplicated code way back earlier when we first showed you the `Person` class, and then later the `Employee` class. Each method used to access the hash fields looked virtually identical. This should have tickled that great programming virtue, Impatience, but for the time, we let Laziness win out, and so did nothing. Proxy methods can cure this.

Instead of writing a new function every time we want a new data field, we'll use the autoload mechanism to

generate (actually, mimic) methods on the fly. To verify that we're accessing a valid member, we will check against an `_permitted` (pronounced "under-permitted") field, which is a reference to a file-scoped lexical (like a C file static) hash of permitted fields in this record called `%fields`. Why the underscore? For the same reason as the `_CENSUS` field we once used: as a marker that means "for internal use only".

Here's what the module initialization code and class constructor will look like when taking this approach:

```
package Person;
use Carp;
our $AUTOLOAD; # it's a package global

my %fields = (
    name      => undef,
    age       => undef,
    peers     => undef,
);

sub new {
    my $that = shift;
    my $class = ref($that) || $that;
    my $self = {
        _permitted => \%fields,
        %fields,
    };
    bless $self, $class;
    return $self;
}
```

If we wanted our record to have default values, we could fill those in where current we have `undef` in the `%fields` hash.

Notice how we saved a reference to our class data on the object itself? Remember that it's important to access class data through the object itself instead of having any method reference `%fields` directly, or else you won't have a decent inheritance.

The real magic, though, is going to reside in our proxy method, which will handle all calls to undefined methods for objects of class `Person` (or subclasses of `Person`). It has to be called `AUTOLOAD`. Again, it's all caps because it's called for us implicitly by Perl itself, not by a user directly.

```
sub AUTOLOAD {
    my $self = shift;
    my $type = ref($self)
        or croak "$self is not an object";

    my $name = $AUTOLOAD;
    $name =~ s/.*://; # strip fully-qualified portion

    unless (exists $self->{_permitted}->{$name} ) {
        croak "Can't access '$name' field in class $type";
    }

    if (@_) {
        return $self->{$name} = shift;
    } else {
        return $self->{$name};
    }
}
```

Pretty nifty, eh? All we have to do to add new data fields is modify `%fields`. No new functions need be written.

I could have avoided the `_permitted` field entirely, but I wanted to demonstrate how to store a reference to class data on the object so you wouldn't have to access that class data directly from an object method.

Inherited Autoloaded Data Methods

But what about inheritance? Can we define our `Employee` class similarly? Yes, so long as we're careful enough.

Here's how to be careful:

```
package Employee;
use Person;
use strict;
our @ISA = qw(Person);

my %fields = (
    id          => undef,
    salary      => undef,
);

sub new {
    my $that = shift;
    my $class = ref($that) || $that;
    my $self = bless $that->SUPER::new(), $class;
    my($element);
    foreach $element (keys %fields) {
        $self->{_permitted}->{$element} = $fields{$element};
    }
    @{$self}{keys %fields} = values %fields;
    return $self;
}
```

Once we've done this, we don't even need to have an `AUTOLOAD` function in the `Employee` package, because we'll grab `Person`'s version of that via inheritance, and it will all work out just fine.

Metaclassical Tools

Even though proxy methods can provide a more convenient approach to making more struct-like classes than tediously coding up data methods as functions, it still leaves a bit to be desired. For one thing, it means you have to handle bogus calls that you don't mean to trap via your proxy. It also means you have to be quite careful when dealing with inheritance, as detailed above.

Perl programmers have responded to this by creating several different class construction classes. These metaclasses are classes that create other classes. A couple worth looking at are `Class::Struct` and `Alias`. These and other related metaclasses can be found in the modules directory on CPAN.

Class::Struct

One of the older ones is `Class::Struct`. In fact, its syntax and interface were sketched out long before perl5 even solidified into a real thing. What it does is provide you a way to "declare" a class as having objects whose fields are of a specific type. The function that does this is called, not surprisingly enough, `struct()`. Because structures or records are not base types in Perl, each time you want to create a class to provide a record-like data object, you yourself have to define a `new()` method, plus separate data-access methods for each of that record's fields. You'll quickly become bored with this process. The `Class::Struct::struct()` function alleviates this tedium.

Here's a simple example of using it:

```
use Class::Struct qw(struct);
use Jobbie; # user-defined; see below

struct 'Fred' => {
    one      => '$',
```

```

        many      => '@',
        profession => Jobbie, # calls Jobbie->new()
    };

    $ob = Fred->new;
    $ob->one("hmmmm");

    $ob->many(0, "here");
    $ob->many(1, "you");
    $ob->many(2, "go");
    print "Just set: ", $ob->many(2), "\n";

    $ob->profession->salary(10_000);

```

You can declare types in the struct to be basic Perl types, or user-defined types (classes). User types will be initialized by calling that class's `new()` method.

Here's a real-world example of using struct generation. Let's say you wanted to override Perl's idea of `gethostbyname()` and `gethostbyaddr()` so that they would return objects that acted like C structures. We don't care about high-falutin' OO gunk. All we want is for these objects to act like structs in the C sense.

```

use Socket;
use Net::hostent;
$h = gethostbyname("perl.com"); # object return
printf "perl.com's real name is %s, address %s\n",
    $h->name, inet_ntoa($h->addr);

```

Here's how to do this using the `Class::Struct` module. The crux is going to be this call:

```

struct 'Net::hostent' => [          # note bracket
    name      => '$',
    aliases   => '@',
    addrtype  => '$',
    'length'  => '$',
    addr_list => '@',
];

```

Which creates object methods of those names and types. It even creates a `new()` method for us.

We could also have implemented our object this way:

```

struct 'Net::hostent' => {          # note brace
    name      => '$',
    aliases   => '@',
    addrtype  => '$',
    'length'  => '$',
    addr_list => '@',
};

```

and then `Class::Struct` would have used an anonymous hash as the object type, instead of an anonymous array. The array is faster and smaller, but the hash works out better if you eventually want to do inheritance. Since for this struct-like object we aren't planning on inheritance, this time we'll opt for better speed and size over better flexibility.

Here's the whole implementation:

```

package Net::hostent;
use strict;

BEGIN {
    use Exporter ();

```

```

our @EXPORT      = qw(gethostbyname gethostbyaddr gethost);
our @EXPORT_OK   = qw(
                        $h_name          @h_aliases
                        $h_addrtype      $h_length
                        @h_addr_list     $h_addr
                    );
our %EXPORT_TAGS = ( FIELDS => [ @EXPORT_OK, @EXPORT ] );
}
our @EXPORT_OK;

# Class::Struct forbids use of @ISA
sub import { goto &Exporter::import }

use Class::Struct qw(struct);
struct 'Net::hostent' => [
    name          => '$',
    aliases        => '@',
    addrtype       => '$',
    'length'       => '$',
    addr_list      => '@',
];

sub addr { shift->addr_list->[0] }

sub populate (@) {
    return unless @_ ;
    my $hob = new(); # Class::Struct made this!
    $h_name      = $hob->[0] = $_[0];
    @h_aliases   = @{ $hob->[1] } = split ' ', $_[1];
    $h_addrtype  = $hob->[2] = $_[2];
    $h_length    = $hob->[3] = $_[3];
    $h_addr      = $hob->[4] = $_[4];
    @h_addr_list = @{ $hob->[4] } = @_[ 4 .. $#_ ];
    return $hob;
}

sub gethostbyname ($) { populate(CORE::gethostbyname(shift)) }

sub gethostbyaddr ($;$) {
    my ($addr, $addrtype);
    $addr = shift;
    require Socket unless @_ ;
    $addrtype = @_ ? shift : Socket::AF_INET();
    populate(CORE::gethostbyaddr($addr, $addrtype))
}

sub gethost($) {
    if ($_[0] =~ /^\\d+(?:\\.\\d+(?:\\.\\d+(?:\\.\\d+)?)?)?$/) {
        require Socket;
        &gethostbyaddr(Socket::inet_aton(shift));
    } else {
        &gethostbyname;
    }
}

1;

```

We've snuck in quite a fair bit of other concepts besides just dynamic class creation, like overriding core functions, import/export bits, function prototyping, short-cut function call via &whatever, and function

replacement with `goto &whatever`. These all mostly make sense from the perspective of a traditional module, but as you can see, we can also use them in an object module.

You can look at other object-based, struct-like overrides of core functions in the 5.004 release of Perl in `File::stat`, `Net::hostent`, `Net::netent`, `Net::protoent`, `Net::servent`, `Time::gmtime`, `Time::localtime`, `User::grent`, and `User::pwent`. These modules have a final component that's all lowercase, by convention reserved for compiler pragmas, because they affect the compilation and change a builtin function. They also have the type names that a C programmer would most expect.

Data Members as Variables

If you're used to C++ objects, then you're accustomed to being able to get at an object's data members as simple variables from within a method. The `Alias` module provides for this, as well as a good bit more, such as the possibility of private methods that the object can call but folks outside the class cannot.

Here's an example of creating a `Person` using the `Alias` module. When you update these magical instance variables, you automatically update value fields in the hash. Convenient, eh?

```
package Person;

# this is the same as before...
sub new {
    my $that = shift;
    my $class = ref($that) || $that;
    my $self = {
        NAME => undef,
        AGE  => undef,
        PEERS => [],
    };
    bless($self, $class);
    return $self;
}

use Alias qw(attr);
our ($NAME, $AGE, $PEERS);

sub name {
    my $self = attr shift;
    if (@_) { $NAME = shift; }
    return $NAME;
}

sub age {
    my $self = attr shift;
    if (@_) { $AGE = shift; }
    return $AGE;
}

sub peers {
    my $self = attr shift;
    if (@_) { @PEERS = @_; }
    return @PEERS;
}

sub exclaim {
    my $self = attr shift;
    return sprintf "Hi, I'm %s, age %d, working with %s",
        $NAME, $AGE, join(", ", @PEERS);
}
```

```

sub happy_birthday {
    my $self = attr shift;
    return ++$AGE;
}

```

The need for the `our` declaration is because what `Alias` does is play with package globals with the same name as the fields. To use globals while `use strict` is in effect, you have to predeclare them. These package variables are localized to the block enclosing the `attr()` call just as if you'd used a `local()` on them. However, that means that they're still considered global variables with temporary values, just as with any other `local()`.

It would be nice to combine `Alias` with something like `Class::Struct` or `Class::MethodMaker`.

NOTES

Object Terminology

In the various OO literature, it seems that a lot of different words are used to describe only a few different concepts. If you're not already an object programmer, then you don't need to worry about all these fancy words. But if you are, then you might like to know how to get at the same concepts in Perl.

For example, it's common to call an object an *instance* of a class and to call those objects' methods *instance methods*. Data fields peculiar to each object are often called *instance data* or *object attributes*, and data fields common to all members of that class are *class data*, *class attributes*, or *static data members*.

Also, *base class*, *generic class*, and *superclass* all describe the same notion, whereas *derived class*, *specific class*, and *subclass* describe the other related one.

C++ programmers have *static methods* and *virtual methods*, but Perl only has *class methods* and *object methods*. Actually, Perl only has methods. Whether a method gets used as a class or object method is by usage only. You could accidentally call a class method (one expecting a string argument) on an object (one expecting a reference), or vice versa.

From the C++ perspective, all methods in Perl are virtual. This, by the way, is why they are never checked for function prototypes in the argument list as regular builtin and user-defined functions can be.

Because a class is itself something of an object, Perl's classes can be taken as describing both a "class as meta-object" (also called *object factory*) philosophy and the "class as type definition" (*declaring* behaviour, not *defining* mechanism) idea. C++ supports the latter notion, but not the former.

SEE ALSO

The following manpages will doubtless provide more background for this one: [perlmod](#), [perlref](#), [perlobj](#), [perlbot](#), [perltie](#), and [overload](#).

[perlboot](#) is a kinder, gentler introduction to object-oriented programming.

[perltoc](#) provides more detail on class data.

Some modules which might prove interesting are `Class::Accessor`, `Class::Class`, `Class::Contract`, `Class::Data::Inheritable`, `Class::MethodMaker` and `Tie::SecureHash`.

AUTHOR AND COPYRIGHT

Copyright (c) 1997, 1998 Tom Christiansen All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

COPYRIGHT**Acknowledgments**

Thanks to Larry Wall, Roderick Schertler, Gurusamy Sarathy, Dean Roehrich, Raphael Manfredi, Brent Halsey, Greg Bacon, Brad Appleton, and many others for their helpful comments.

NAME

perltrap – Perl traps for the unwary

DESCRIPTION

The biggest trap of all is forgetting to use `warnings` or use the `-w` switch; see [perllexwarn](#) and [perlrun](#). The second biggest trap is not making your entire program runnable under `use strict`. The third biggest trap is not reading the list of changes in this version of Perl; see [perldelta](#).

Awk Traps

Accustomed **awk** users should take special note of the following:

- The English module, loaded via


```
use English;
```

 allows you to refer to special variables (like `$/`) with names (like `$RS`), as though they were in **awk**; see [perlvar](#) for details.
- Semicolons are required after all simple statements in Perl (except at the end of a block). Newline is not a statement delimiter.
- Curly brackets are required on `ifs` and `whiles`.
- Variables begin with `"$"`, `"@"` or `"%"` in Perl.
- Arrays index from 0. Likewise string positions in `substr()` and `index()`.
- You have to decide whether your array has numeric or string indices.
- Hash values do not spring into existence upon mere reference.
- You have to decide whether you want to use string or numeric comparisons.
- Reading an input line does not split it for you. You get to split it to an array yourself. And the `split()` operator has different arguments than **awk**'s.
- The current input line is normally in `$_`, not `$0`. It generally does not have the newline stripped. (`$0` is the name of the program executed.) See [perlvar](#).
- `$<digit` does not refer to fields—it refers to substrings matched by the last match pattern.
- The `print()` statement does not add field and record separators unless you set `$,` and `$\`. You can set `$OFS` and `$ORS` if you're using the English module.
- You must open your files before you print to them.
- The range operator is `".."`, not comma. The comma operator works as in C.
- The match operator is `"=~"`, not `"~"`. (`"~"` is the one's complement operator, as in C.)
- The exponentiation operator is `"**"`, not `"^"`. `"^"` is the XOR operator, as in C. (You know, one could get the feeling that **awk** is basically incompatible with C.)
- The concatenation operator is `"."`, not the null string. (Using the null string would render `/pat/` `/pat/` unparseable, because the third slash would be interpreted as a division operator—the tokenizer is in fact slightly context sensitive for operators like `"/"`, `"?"`, and `"."`. And in fact, `"."` itself can be the beginning of a number.)
- The `next`, `exit`, and `continue` keywords work differently.
- The following variables work differently:

Awk	Perl
ARGC	scalar @ARGV (compare with \$#ARGV)
ARGV[0]	\$0

```

FILENAME  $ARGV
FN$. - something
FS(whatever you like)
NF$#Fld, or some such
NR$.
OF$#
OF$,
OR$\
RLENGTH  length($&)
RS$/
RSTART    length($`)
SUBSEP    $;

```

- You cannot set `$RS` to a pattern, only a string.
- When in doubt, run the **awk** construct through **a2p** and see what it gives you.

C Traps

Cerebral C programmers should take note of the following:

- Curly brackets are required on `if`'s and `while`'s.
- You must use `elsif` rather than `else if`.
- The `break` and `continue` keywords from C become in Perl `last` and `next`, respectively. Unlike in C, these do *not* work within a `do { } while` construct.
- There's no `switch` statement. (But it's easy to build one on the fly.)
- Variables begin with "\$", "@", or "%" in Perl.
- Comments begin with "#", not "/*".
- You can't take the address of anything, although a similar operator in Perl is the backslash, which creates a reference.
- `ARGV` must be capitalized. `$ARGV[0]` is C's `argv[1]`, and `argv[0]` ends up in `$0`.
- System calls such as `link()`, `unlink()`, `rename()`, etc. return nonzero for success, not 0. (`system()`, however, returns zero for success.)
- Signal handlers deal with signal names, not numbers. Use `kill -1` to find their names on your system.

Sed Traps

Seasoned **sed** programmers should take note of the following:

- Backreferences in substitutions use "\$" rather than "\".
- The pattern matching metacharacters "(", ")", and "|" do not have backslashes in front.
- The range operator is `...` , rather than comma.

Shell Traps

Sharp shell programmers should take note of the following:

- The backtick operator does variable interpolation without regard to the presence of single quotes in the command.
- The backtick operator does no translation of the return value, unlike **csh**.
- Shells (especially **csh**) do several levels of substitution on each command line. Perl does substitution in only certain constructs such as double quotes, backticks, angle brackets, and search patterns.

- Shells interpret scripts a little bit at a time. Perl compiles the entire program before executing it (except for BEGIN blocks, which execute at compile time).
- The arguments are available via @ARGV, not \$1, \$2, etc.
- The environment is not automatically made available as separate scalar variables.

Perl Traps

Practicing Perl Programmers should take note of the following:

- Remember that many operations behave differently in a list context than they do in a scalar one. See [perldata](#) for details.
- Avoid barewords if you can, especially all lowercase ones. You can't tell by just looking at it whether a bareword is a function or a string. By using quotes on strings and parentheses on function calls, you won't ever get them confused.
- You cannot discern from mere inspection which builtins are unary operators (like `chop()` and `chdir()`) and which are list operators (like `print()` and `unlink()`). (Unless prototyped, user-defined subroutines can **only** be list operators, never unary ones.) See [perlop](#) and [perlsub](#).
- People have a hard time remembering that some functions default to `$_`, or @ARGV, or whatever, but that others which you might expect to do not.
- The `<FH` construct is not the name of the filehandle, it is a readline operation on that handle. The data read is assigned to `$_` only if the file read is the sole condition in a while loop:

```
while (<FH>)      { }
while (defined($_ = <FH>)) { }..
<FH>; # data discarded!
```

- Remember not to use `=` when you need `=~`; these two constructs are quite different:

```
$x = /foo/;
$x =~ /foo/;
```

- The `do { }` construct isn't a real loop that you can use loop control on.
- Use `my()` for local variables whenever you can get away with it (but see [perlform](#) for where you can't). Using `local()` actually gives a local value to a global variable, which leaves you open to unforeseen side-effects of dynamic scoping.
- If you localize an exported variable in a module, its exported value will not change. The local name becomes an alias to a new value but the external name is still an alias for the original.

Perl4 to Perl5 Traps

Practicing Perl4 Programmers should take note of the following Perl4-to-Perl5 specific traps.

They're crudely ordered according to the following list:

Discontinuance, Deprecation, and BugFix traps

Anything that's been fixed as a perl4 bug, removed as a perl4 feature or deprecated as a perl4 feature with the intent to encourage usage of some other perl5 feature.

Parsing Traps

Traps that appear to stem from the new parser.

Numerical Traps

Traps having to do with numerical or mathematical operators.

General data type traps

Traps involving perl standard data types.

Context Traps – scalar, list contexts

Traps related to context within lists, scalar statements/declarations.

Precedence Traps

Traps related to the precedence of parsing, evaluation, and execution of code.

General Regular Expression Traps using s///, etc.

Traps related to the use of pattern matching.

Subroutine, Signal, Sorting Traps

Traps related to the use of signals and signal handlers, general subroutines, and sorting, along with sorting subroutines.

OS Traps

OS-specific traps.

DBM Traps

Traps specific to the use of `dbmopen()`, and specific dbm implementations.

Unclassified Traps

Everything else.

If you find an example of a conversion trap that is not listed here, please submit it to [<perlbug@perl.org>](mailto:perlbug@perl.org) for inclusion. Also note that at least some of these can be caught with the use `warnings` pragma or the `-w` switch.

Discontinuance, Deprecation, and BugFix traps

Anything that has been discontinued, deprecated, or fixed as a bug from perl4.

- **Discontinuance**

Symbols starting with "_" are no longer forced into package main, except for `$_` itself (and `@_`, etc.).

```
package test;
$_legacy = 1;

package main;
print "\$_legacy is ", $_legacy, "\n";

# perl4 prints: $_legacy is 1
# perl5 prints: $_legacy is
```

- **Deprecation**

Double-colon is now a valid package separator in a variable name. Thus these behave differently in perl4 vs. perl5, because the packages don't exist.

```
$a=1;$b=2;$c=3;$var=4;
print "$a::$b::$c ";
print "$var::abc::xyz\n";

# perl4 prints: 1::2::3 4::abc::xyz
# perl5 prints: 3
```

Given that `::` is now the preferred package delimiter, it is debatable whether this should be classed as a bug or not. (The older package delimiter, `'`, is used here)

```
$x = 10 ;
print "x=${'x'}\n" ;

# perl4 prints: x=10
# perl5 prints: Can't find string terminator "'" anywhere before EOF
```

You can avoid this problem, and remain compatible with perl4, if you always explicitly include the package name:

```
$x = 10 ;
print "x=${main'x'}\n" ;
```

Also see precedence traps, for parsing \$: .

• BugFix

The second and third arguments of `splice()` are now evaluated in scalar context (as the Camel says) rather than list context.

```
sub sub1{return(0,2) }           # return a 2-element list
sub sub2{ return(1,2,3) }       # return a 3-element list
@a1 = ("a","b","c","d","e");
@a2 = splice(@a1,&sub1,&sub2);
print join(' ',@a2),"\n";

# perl4 prints: a b
# perl5 prints: c d e
```

• Discontinuance

You can't do a `goto` into a block that is optimized away. Darn.

```
goto marker1;

for(1){
marker1:
    print "Here I is!\n";
}

# perl4 prints: Here I is!
# perl5 errors: Can't "goto" into the middle of a foreach loop
```

• Discontinuance

It is no longer syntactically legal to use whitespace as the name of a variable, or as a delimiter for any kind of quote construct. Double darn.

```
$a = ("foo bar");
$b = q baz ;
print "a is $a, b is $b\n";

# perl4 prints: a is foo bar, b is baz
# perl5 errors: Bareword found where operator expected
```

• Discontinuance

The archaic `while/if BLOCK BLOCK` syntax is no longer supported.

```
if { 1 } {
    print "True!";
}
else {
    print "False!";
}

# perl4 prints: True!
# perl5 errors: syntax error at test.pl line 1, near "if {"
```

• BugFix

The `**` operator now binds more tightly than unary minus. It was documented to work this way before, but didn't.

```
print -4**2, "\n";

# perl4 prints: 16
# perl5 prints: -16
```

• Discontinuance

The meaning of `foreach{ }` has changed slightly when it is iterating over a list which is not an array. This used to assign the list to a temporary array, but no longer does so (for efficiency). This means that you'll now be iterating over the actual values, not over copies of the values. Modifications to the loop variable can change the original values.

```
@list = ('ab', 'abc', 'bcd', 'def');
foreach $var (grep(/ab/, @list)) {
    $var = 1;
}
print (join(':', @list));

# perl4 prints: ab:abc:bcd:def
# perl5 prints: 1:1:bcd:def
```

To retain Perl4 semantics you need to assign your list explicitly to a temporary array and then iterate over that. For example, you might need to change

```
foreach $var (grep(/ab/, @list)) {
```

to

```
foreach $var (@tmp = grep(/ab/, @list)) {
```

Otherwise changing `$var` will clobber the values of `@list`. (This most often happens when you use `$_` for the loop variable, and call subroutines in the loop that don't properly localize `$_`.)

• Discontinuance

`split` with no arguments now behaves like `split ' '` (which doesn't return an initial null field if `$_` starts with whitespace), it used to behave like `split /\s+/` (which does).

```
$_ = ' hi mom';
print join(':', split);

# perl4 prints: :hi:mom
# perl5 prints: hi:mom
```

• BugFix

Perl 4 would ignore any text which was attached to an `-e` switch, always taking the code snippet from the following arg. Additionally, it would silently accept an `-e` switch without a following arg. Both of these behaviors have been fixed.

```
perl -e'print "attached to -e"' 'print "separate arg"'

# perl4 prints: separate arg
# perl5 prints: attached to -e

perl -e

# perl4 prints:
# perl5 dies: No code specified for -e.
```

• Discontinuance

In Perl 4 the return value of `push` was undocumented, but it was actually the last value being pushed onto the target list. In Perl 5 the return value of `push` is documented, but has changed, it is the number of elements in the resulting list.

```
@x = ('existing');
```

```
print push(@x, 'first new', 'second new');
# perl4 prints: second new
# perl5 prints: 3
```

- Deprecation

Some error messages will be different.

- Discontinuance

In Perl 4, if in list context the delimiters to the first argument of `split()` were `??`, the result would be placed in `@_` as well as being returned. Perl 5 has more respect for your subroutine arguments.

- Discontinuance

Some bugs may have been inadvertently removed. :-)

Parsing Traps

Perl4-to-Perl5 traps from having to do with parsing.

- Parsing

Note the space between `.` and `=`

```
$string . = "more string";
print $string;

# perl4 prints: more string
# perl5 prints: syntax error at - line 1, near ". ="
```

- Parsing

Better parsing in perl 5

```
sub foo {}
&foo
print("hello, world\n");

# perl4 prints: hello, world
# perl5 prints: syntax error
```

- Parsing

"if it looks like a function, it is a function" rule.

```
print
($foo == 1) ? "is one\n" : "is zero\n";

# perl4 prints: is zero
# perl5 warns: "Useless use of a constant in void context" if using -w
```

- Parsing

String interpolation of the `$#array` construct differs when braces are to used around the name.

```
@a = (1..3);
print "${#a}";

# perl4 prints: 2
# perl5 fails with syntax error

@ = (1..3);
print "$#{a}";

# perl4 prints: {a}
# perl5 prints: 2
```


- Parsing

When perl sees `map {` (or `grep {`), it has to guess whether the `{` starts a BLOCK or a hash reference. If it guesses wrong, it will report a syntax error near the `}` and the missing (or unexpected) comma.

Use unary `+` before `{` on a hash reference, and unary `+` applied to the first thing in a BLOCK (after `{`), for perl to guess right all the time. (See [map](#).)

Numerical Traps

Perl4-to-Perl5 traps having to do with numerical operators, operands, or output from same.

- Numerical

Formatted output and significant digits. In general, Perl 5 tries to be more precise. For example, on a Solaris Sparc:

```
print 7.373504 - 0, "\n";
printf "%20.18f\n", 7.373504 - 0;

# Perl4 prints:
7.3750399999999996141
7.375039999999999614

# Perl5 prints:
7.373504
7.375039999999999614
```

Notice how the first result looks better in Perl 5.

Your results may vary, since your floating point formatting routines and even floating point format may be slightly different.

- Numerical

This specific item has been deleted. It demonstrated how the auto-increment operator would not catch when a number went over the signed int limit. Fixed in version 5.003_04. But always be wary when using large integers. If in doubt:

```
use Math::BigInt;
```

- Numerical

Assignment of return values from numeric equality tests does not work in perl5 when the test evaluates to false (0). Logical tests now return a null, instead of 0

```
$p = ($test == 1);
print $p, "\n";

# perl4 prints: 0
# perl5 prints:
```

Also see [, etc.](#) for another example of this new feature...

- Bitwise string ops

When bitwise operators which can operate upon either numbers or strings (`&` `|` `^` `~`) are given only strings as arguments, perl4 would treat the operands as bitstrings so long as the program contained a call to the `vec()` function. perl5 treats the string operands as bitstrings. (See [Bitwise String Operators](#) for more details.)

```
$fred = "10";
$barney = "12";
$betty = $fred & $barney;
print "$betty\n";
# Uncomment the next line to change perl4's behavior
```

```
# ($dummy) = vec("dummy", 0, 0);
# Perl4 prints:
8
# Perl5 prints:
10
# If vec() is used anywhere in the program, both print:
10
```

General data type traps

Perl4-to-Perl5 traps involving most data-types, and their usage within certain expressions and/or context.

- (Arrays)

Negative array subscripts now count from the end of the array.

```
@a = (1, 2, 3, 4, 5);
print "The third element of the array is $a[3] also expressed as $a[-2] \n";

# perl4 prints: The third element of the array is 4 also expressed as
# perl5 prints: The third element of the array is 4 also expressed as 4
```

- (Arrays)

Setting \$#array lower now discards array elements, and makes them impossible to recover.

```
@a = (a,b,c,d,e);
print "Before: ",join(' ',@a);
$a# =1;
print ", After: ",join(' ',@a);
$a# =3;
print ", Recovered: ",join(' ',@a),"\n";

# perl4 prints: Before: abcde, After: ab, Recovered: abcd
# perl5 prints: Before: abcde, After: ab, Recovered: ab
```

- (Hashes)

Hashes get defined before use

```
local($s,@a,%h);
die "scalar \$s defined" if defined($s);
die "array \@a defined" if defined(@a);
die "hash \%h defined" if defined(%h);

# perl4 prints:
# perl5 dies: hash %h defined
```

Perl will now generate a warning when it sees defined(@a) and defined(%h).

- (Globs)

glob assignment from variable to variable will fail if the assigned variable is localized subsequent to the assignment

```
@a = ("This is Perl 4");
*b = *a;
local(@a);
print @b,"\n";

# perl4 prints: This is Perl 4
# perl5 prints:
```

- (Globs)

Assigning undef to a glob has no effect in Perl 5. In Perl 4 it undefines the associated scalar (but may have other side effects including SEGVs). Perl 5 will also warn if undef is assigned to a typeglob. (Note that assigning undef to a typeglob is different than calling the undef function on a typeglob (undef *foo), which has quite a few effects.

```
$foo = "bar";
*foo = undef;
print $foo;

# perl4 prints:
# perl4 warns: "Use of uninitialized variable" if using -w
# perl5 prints: bar
# perl5 warns: "Undefined value assigned to typeglob" if using -w
```

- (Scalar String)

Changes in unary negation (of strings) This change effects both the return value and what it does to auto(magic)increment.

```
$x = "aaa";
print ++$x, " : ";
print -$x, " : ";
print ++$x, "\n";

# perl4 prints: aab : -0 : 1
# perl5 prints: aab : -aab : aac
```

- (Constants)

perl 4 lets you modify constants:

```
$foo = "x";
&mod($foo);
for ($x = 0; $x < 3; $x++) {
    &mod("a");
}
sub mod {
    print "before: $_[0]";
    $_[0] = "m";
    print "  after: $_[0]\n";
}

# perl4:
# before: x  after: m
# before: a  after: m
# before: m  after: m
# before: m  after: m

# Perl5:
# before: x  after: m
# Modification of a read-only value attempted at foo.pl line 12.
# before: a
```

- (Scalars)

The behavior is slightly different for:

```
print "$x", defined $x

# perl 4: 1
# perl 5: <no output, $x is not called into existence>
```

- (Variable Suicide)

Variable suicide behavior is more consistent under Perl 5. Perl5 exhibits the same behavior for hashes and scalars, that perl4 exhibits for only scalars.

```
$aGlobal{ "aKey" } = "global value";
print "MAIN:", $aGlobal{"aKey"}, "\n";
$GlobalLevel = 0;
&test( *aGlobal );

sub test {
    local( *theArgument ) = @_;
    local( %aNewLocal ); # perl 4 != 5.0011,m
    $aNewLocal{"aKey"} = "this should never appear";
    print "SUB: ", $theArgument{"aKey"}, "\n";
    $aNewLocal{"aKey"} = "level $GlobalLevel";    # what should print
    $GlobalLevel++;
    if( $GlobalLevel<4 ) {
        &test( *aNewLocal );
    }
}

# Perl4:
# MAIN:global value
# SUB: global value
# SUB: level 0
# SUB: level 1
# SUB: level 2

# Perl5:
# MAIN:global value
# SUB: global value
# SUB: this should never appear
# SUB: this should never appear
# SUB: this should never appear
```

Context Traps – scalar, list contexts

- (list context)

The elements of argument lists for formats are now evaluated in list context. This means you can interpolate list values now.

```
@fmt = ("foo", "bar", "baz");
format STDOUT=
@<<<<< @||| | @>>>>>
@fmt;

.
write;

# perl4 errors: Please use commas to separate fields in file
# perl5 prints: foo      bar      baz
```

- (scalar context)

The caller() function now returns a false value in a scalar context if there is no caller. This lets library files determine if they're being required.

```
caller() ? (print "You rang?\n") : (print "Got a 0\n");

# perl4 errors: There is no caller
# perl5 prints: Got a 0
```

- (scalar context)

The comma operator in a scalar context is now guaranteed to give a scalar context to its arguments.

```
@y= ('a', 'b', 'c');
$x = (1, 2, @y);
print "x = $x\n";

# Perl4 prints: x = c    # Thinks list context interpolates list
# Perl5 prints: x = 3    # Knows scalar uses length of list
```

- (list, builtin)

`sprintf()` is prototyped as `($;@)`, so its first argument is given scalar context. Thus, if passed an array, it will probably not do what you want, unlike Perl 4:

```
@z = ('%s%s', 'foo', 'bar');
$x = sprintf(@z);
print $x;

# perl4 prints: foobar
# perl5 prints: 3
```

`printf()` works the same as it did in Perl 4, though:

```
@z = ('%s%s', 'foo', 'bar');
printf STDOUT (@z);

# perl4 prints: foobar
# perl5 prints: foobar
```

Precedence Traps

Perl4-to-Perl5 traps involving precedence order.

Perl 4 has almost the same precedence rules as Perl 5 for the operators that they both have. Perl 4 however, seems to have had some inconsistencies that made the behavior differ from what was documented.

- Precedence

LHS vs. RHS of any assignment operator. LHS is evaluated first in perl4, second in perl5; this can affect the relationship between side-effects in sub-expressions.

```
@arr = ( 'left', 'right' );
${shift @arr} = shift @arr;
print join( ' ', keys %a );

# perl4 prints: left
# perl5 prints: right
```

- Precedence

These are now semantic errors because of precedence:

```
@list = (1,2,3,4,5);
%map = ("a",1,"b",2,"c",3,"d",4);
$n = shift @list + 2;    # first item in list plus 2
print "n is $n, ";
$m = keys %map + 2;      # number of items in hash plus 2
print "m is $m\n";

# perl4 prints: n is 3, m is 6
# perl5 errors and fails to compile
```

- Precedence

The precedence of assignment operators is now the same as the precedence of assignment. Perl 4 mistakenly gave them the precedence of the associated operator. So you now must parenthesize them in expressions like

```
/foo/ ? ($a += 2) : ($a -= 2);
```

Otherwise

```
/foo/ ? $a += 2 : $a -= 2
```

would be erroneously parsed as

```
(/foo/ ? $a += 2 : $a) -= 2;
```

On the other hand,

```
$a += /foo/ ? 1 : 2;
```

now works as a C programmer would expect.

- Precedence

```
open FOO || die;
```

is now incorrect. You need parentheses around the filehandle. Otherwise, perl5 leaves the statement as its default precedence:

```
open(FOO || die);
# perl4 opens or dies
# perl5 opens FOO, dying only if 'FOO' is false, i.e. never
```

- Precedence

perl4 gives the special variable, \$: precedence, where perl5 treats \$: as main package

```
$a = "x"; print "$::a";
# perl 4 prints: -:a
# perl 5 prints: x
```

- Precedence

perl4 had buggy precedence for the file test operators vis-a-vis the assignment operators. Thus, although the precedence table for perl4 leads one to believe `-e $foo .= "q"` should parse as `((-e $foo) .= "q")`, it actually parses as `(-e ($foo .= "q"))`. In perl5, the precedence is as documented.

```
-e $foo .= "q"
# perl4 prints: no output
# perl5 prints: Can't modify -e in concatenation
```

- Precedence

In perl4, `keys()`, `each()` and `values()` were special high-precedence operators that operated on a single hash, but in perl5, they are regular named unary operators. As documented, named unary operators have lower precedence than the arithmetic and concatenation operators `+` `-` `..`, but the perl4 variants of these operators actually bind tighter than `+` `-` `..`. Thus, for:

```
%foo = 1..10;
print keys %foo - 1
# perl4 prints: 4
# perl5 prints: Type of arg 1 to keys must be hash (not subtraction)
```

The perl4 behavior was probably more useful, if less consistent.

General Regular Expression Traps using `s///`, etc.

All types of RE traps.

• Regular Expression

`s`$lhs`$rhs`` now does no interpolation on either side. It used to interpolate `$lhs` but not `$rhs`. (And still does not match a literal `'$'` in string)

```
$a=1;$b=2;
$string = '1 2 $a $b';
$string =~ s'$a'$b';
print $string, "\n";

# perl4 prints: $b 2 $a $b
# perl5 prints: 1 2 $a $b
```

• Regular Expression

`m/g` now attaches its state to the searched string rather than the regular expression. (Once the scope of a block is left for the sub, the state of the searched string is lost)

```
$_ = "ababab";
while(m/ab/g){
    &doit("blah");
}
sub doit{local($_) = shift; print "Got $_ "}

# perl4 prints: Got blah Got blah Got blah Got blah
# perl5 prints: infinite loop blah...
```

• Regular Expression

Currently, if you use the `m/o` qualifier on a regular expression within an anonymous sub, *all* closures generated from that anonymous sub will use the regular expression as it was compiled when it was used the very first time in any such closure. For instance, if you say

```
sub build_match {
    my($left,$right) = @_;
    return sub { $_[0] =~ /$left stuff $right/o; };
}
$good = build_match('foo','bar');
$bad = build_match('baz','blarch');
print $good->('foo stuff bar') ? "ok\n" : "not ok\n";
print $bad->('baz stuff blarch') ? "ok\n" : "not ok\n";
print $bad->('foo stuff bar') ? "not ok\n" : "ok\n";
```

For most builds of Perl5, this will print: ok not ok not ok

`build_match()` will always return a sub which matches the contents of `$left` and `$right` as they were the *first* time that `build_match()` was called, not as they are in the current call.

• Regular Expression

If no parentheses are used in a match, Perl4 sets `$+` to the whole match, just like `$&`. Perl5 does not.

```
"abcdef" =~ /b.*e/;
print "\$+ = $+\n";

# perl4 prints: bcde
# perl5 prints:
```

• Regular Expression

substitution now returns the null string if it fails

```

$string = "test";
$value = ($string =~ s/foo//);
print $value, "\n";

# perl4 prints: 0
# perl5 prints:

```

Also see [Numerical Traps](#) for another example of this new feature.

- Regular Expression

`s`lhs`rhs`` (using backticks) is now a normal substitution, with no backtick expansion

```

$string = "";
$string =~ s/`hostname`;
print $string, "\n";

# perl4 prints: <the local hostname>
# perl5 prints: hostname

```

- Regular Expression

Stricter parsing of variables used in regular expressions

```

s/^( [^$grpc]*$grpc[$opt$plus$rep]? )//o;

# perl4: compiles w/o error
# perl5: with Scalar found where operator expected ..., near "$opt$plus"

```

an added component of this example, apparently from the same script, is the actual value of the `s'd` string after the substitution. `[$opt]` is a character class in perl4 and an array subscript in perl5

```

$grpc = 'a';
$opt = 'r';
$_ = 'bar';
s/^( [^$grpc]*$grpc[$opt]? )/foo/;
print ;

# perl4 prints: foo
# perl5 prints: foobar

```

- Regular Expression

Under perl5, `m?x?` matches only once, like `?x?`. Under perl4, it matched repeatedly, like `/x/` or `m!x!`.

```

$test = "once";
sub match { $test =~ m?once?; }
&match();
if( &match() ) {
    # m?x? matches more than once
    print "perl4\n";
} else {
    # m?x? matches only once
    print "perl5\n";
}

# perl4 prints: perl4
# perl5 prints: perl5

```

Subroutine, Signal, Sorting Traps

The general group of Perl4-to-Perl5 traps having to do with Signals, Sorting, and their related subroutines, as well as general subroutine traps. Includes some OS-Specific traps.

- (Signals)

Barewords that used to look like strings to Perl will now look like subroutine calls if a subroutine by that name is defined before the compiler sees them.

```
sub SeeYa { warn"Hasta la vista, baby!" }
$SIG{'TERM'} = SeeYa;
print "SIGTERM is now $SIG{'TERM'}\n";

# perl4 prints: SIGTERM is now main'SeeYa
# perl5 prints: SIGTERM is now main::1 (and warns "Hasta la vista, baby!")
```

Use `-w` to catch this one

- (Sort Subroutine)

`reverse` is no longer allowed as the name of a sort subroutine.

```
sub reverse{ print "yup "; $a <=> $b }
print sort reverse (2,1,3);

# perl4 prints: yup yup 123
# perl5 prints: 123
# perl5 warns (if using -w): Ambiguous call resolved as CORE::reverse()
```

- `warn()` won't let you specify a filehandle.

Although it `_always_` printed to `STDERR`, `warn()` would let you specify a filehandle in perl4. With perl5 it does not.

```
warn STDERR "Foo!";

# perl4 prints: Foo!
# perl5 prints: String found where operator expected
```

OS Traps

- (SysV)

Under HP/UX, and some other SysV OSes, one had to reset any signal handler, within the signal handler function, each time a signal was handled with perl4. With perl5, the reset is now done correctly. Any code relying on the handler `_not_` being reset will have to be reworked.

Since version 5.002, Perl uses `sigaction()` under SysV.

```
sub gotit {
    print "Got @_... ";
}
$SIG{'INT'} = 'gotit';

$| = 1;
$pid = fork;
if ($pid) {
    kill('INT', $pid);
    sleep(1);
    kill('INT', $pid);
} else {
    while (1) {sleep(10);}
}

# perl4 (HP/UX) prints: Got INT...
# perl5 (HP/UX) prints: Got INT... Got INT...
```

- (SysV)

Under SysV OSes, `seek()` on a file opened to append << now does the right thing w.r.t. the `fopen()` manpage. e.g., – When a file is opened for append, it is impossible to overwrite information already in the file.

```
open( TEST, ">>seek.test" );
$start = tell TEST ;
foreach( 1 .. 9 ) {
    print TEST "$_ ";
}
$end = tell TEST ;
seek( TEST, $start, 0 );
print TEST "18 characters here";

# perl4 (solaris) seek.test has: 18 characters here
# perl5 (solaris) seek.test has: 1 2 3 4 5 6 7 8 9 18 characters here
```

Interpolation Traps

Perl4-to-Perl5 traps having to do with how things get interpolated within certain expressions, statements, contexts, or whatever.

- Interpolation

@ now always interpolates an array in double-quotish strings.

```
print "To: someone@somewhere.com\n";

# perl4 prints: To:someone@somewhere.com
# perl < 5.6.1, error : In string, @somewhere now must be written as \@somew
# perl >= 5.6.1, warning : Possible unintended interpolation of @somewhere i
```

- Interpolation

Double-quoted strings may no longer end with an unescaped `$` or `@`.

```
$foo = "foo$";
$bar = "bar@";
print "foo is $foo, bar is $bar\n";

# perl4 prints: foo is foo$, bar is bar@
# perl5 errors: Final $ should be \$ or $name
```

Note: perl5 DOES NOT error on the terminating `@` in `$bar`

- Interpolation

Perl now sometimes evaluates arbitrary expressions inside braces that occur within double quotes (usually when the opening brace is preceded by `$` or `@`).

```
@www = "buz";
$foo = "foo";
$bar = "bar";
sub foo { return "bar" };
print "|@{w.w.w}|${main'foo}|" ;

# perl4 prints: |@{w.w.w}|foo|
# perl5 prints: |buz|bar|
```

Note that you can use `strict`; to ward off such trappiness under perl5.

- Interpolation

The construct "this is `$$x`" used to interpolate the pid at that point, but now tries to dereference `$x`. `$$` by itself still works fine, however.

```

$s = "a reference";
$x = *s;
print "this is $$x\n";

# perl4 prints: this is XXXx   (XXX is the current pid)
# perl5 prints: this is a reference

```

- Interpolation

Creation of hashes on the fly with `eval "EXPR"` now requires either both `$`'s to be protected in the specification of the hash name, or both curlies to be protected. If both curlies are protected, the result will be compatible with perl4 and perl5. This is a very common practice, and should be changed to use the block form of `eval{ }` if possible.

```

$hashname = "foobar";
$key = "baz";
$value = 1234;
eval "\$$hashname{'$key'} = q|$value|";
(defined($foobar{'baz'})) ? (print "Yup") : (print "Nope");

# perl4 prints: Yup
# perl5 prints: Nope

```

Changing

```
eval "\$$hashname{'$key'} = q|$value|";
```

to

```
eval "\$\$hashname{'$key'} = q|$value|";
```

causes the following result:

```

# perl4 prints: Nope
# perl5 prints: Yup

```

or, changing to

```
eval "\$$hashname\{'$key'\} = q|$value|";
```

causes the following result:

```

# perl4 prints: Yup
# perl5 prints: Yup
# and is compatible for both versions

```

- Interpolation

perl4 programs which unconsciously rely on the bugs in earlier perl versions.

```

perl -e '$bar=q/not/; print "This is $foo{$bar} perl5"'

# perl4 prints: This is not perl5
# perl5 prints: This is perl5

```

- Interpolation

You also have to be careful about array references.

```

print "$foo{"
perl 4 prints: {
perl 5 prints: syntax error

```

- Interpolation

Similarly, watch out for:

```
$foo = "baz";
print "\$$foo{bar}\n";

# perl4 prints: $baz{bar}
# perl5 prints: $
```

Perl 5 is looking for `$foo{bar}` which doesn't exist, but perl 4 is happy just to expand `$foo` to "baz" by itself. Watch out for this especially in `eval`'s.

- **Interpolation**

`qq()` string passed to `eval`

```
eval qq(
    foreach \$y (keys %\$x\ ) {
        \$count++;
    }
);

# perl4 runs this ok
# perl5 prints: Can't find string terminator ")"
```

DBM Traps

General DBM traps.

- **DBM** Existing dbm databases created under perl4 (or any other dbm/ndbm tool) may cause the same script, run under perl5, to fail. The build of perl5 must have been linked with the same dbm/ndbm as the default for `dbmopen()` to function properly without tie'ing to an extension dbm implementation.

```
dbmopen (%dbm, "file", undef);
print "ok\n";

# perl4 prints: ok
# perl5 prints: ok (IFF linked with -ldb or -lndbm)
```

- **DBM** Existing dbm databases created under perl4 (or any other dbm/ndbm tool) may cause the same script, run under perl5, to fail. The error generated when exceeding the limit on the key/value size will cause perl5 to exit immediately.

```
dbmopen(DB, "testdb", 0600) || die "couldn't open db! $!";
$DB{'trap'} = "x" x 1024; # value too large for most dbm/ndbm
print "YUP\n";

# perl4 prints:
dbm store returned -1, errno 28, key "trap" at - line 3.
YUP

# perl5 prints:
dbm store returned -1, errno 28, key "trap" at - line 3.
```

Unclassified Traps

Everything else.

- **require/do trap** using returned value

If the file `doit.pl` has:

```
sub foo {
    $rc = do "./do.pl";
    return 8;
}

print &foo, "\n";
```

And the `do.pl` file has the following single line:

```
return 3;
```

Running doit.pl gives the following:

```
# perl 4 prints: 3 (aborts the subroutine early)
# perl 5 prints: 8
```

Same behavior if you replace do with require.

- `split` on empty string with LIMIT specified

```
$string = '';
@list = split(/foo/, $string, 2)
```

Perl4 returns a one element list containing the empty string but Perl5 returns an empty list.

As always, if any of these are ever officially declared as bugs, they'll be fixed and removed.

NAME

perlunicode – Unicode support in Perl

DESCRIPTION**Important Caveats**

Unicode support is an extensive requirement. While Perl does not implement the Unicode standard or the accompanying technical reports from cover to cover, Perl does support many Unicode features.

Input and Output Layers

Perl knows when a filehandle uses Perl's internal Unicode encodings (UTF-8, or UTF-EBCDIC if in EBCDIC) if the filehandle is opened with the ":utf8" layer. Other encodings can be converted to Perl's encoding on input or from Perl's encoding on output by use of the ":encoding(...)" layer. See [open](#).

To indicate that Perl source itself is using a particular encoding, see [encoding](#).

Regular Expressions

The regular expression compiler produces polymorphic opcodes. That is, the pattern adapts to the data and automatically switches to the Unicode character scheme when presented with Unicode data—or instead uses a traditional byte scheme when presented with byte data.

use utf8 still needed to enable UTF-8/UTF-EBCDIC in scripts

As a compatibility measure, the `use utf8` pragma must be explicitly included to enable recognition of UTF-8 in the Perl scripts themselves (in string or regular expression literals, or in identifier names) on ASCII-based machines or to recognize UTF-EBCDIC on EBCDIC-based machines. **These are the only times when an explicit `use utf8` is needed.** See [utf8](#).

You can also use the `encoding` pragma to change the default encoding of the data in your script; see [encoding](#).

Byte and Character Semantics

Beginning with version 5.6, Perl uses logically-wide characters to represent strings internally.

In future, Perl-level operations will be expected to work with characters rather than bytes.

However, as an interim compatibility measure, Perl aims to provide a safe migration path from byte semantics to character semantics for programs. For operations where Perl can unambiguously decide that the input data are characters, Perl switches to character semantics. For operations where this determination cannot be made without additional information from the user, Perl decides in favor of compatibility and chooses to use byte semantics.

This behavior preserves compatibility with earlier versions of Perl, which allowed byte semantics in Perl operations only if none of the program's inputs were marked as being as source of Unicode character data. Such data may come from filehandles, from calls to external programs, from information provided by the system (such as %ENV), or from literals and constants in the source text.

On Windows platforms, if the `-C` command line switch is used or the `$_{^WIDE_SYSTEM_CALLS}` global flag is set to 1, all system calls will use the corresponding wide-character APIs. This feature is available only on Windows to conform to the API standard already established for that platform—and there are very few non-Windows platforms that have Unicode-aware APIs.

The `bytes` pragma will always, regardless of platform, force byte semantics in a particular lexical scope. See [bytes](#).

The `utf8` pragma is primarily a compatibility device that enables recognition of UTF-(8|EBCDIC) in literals encountered by the parser. Note that this pragma is only required while Perl defaults to byte semantics; when character semantics become the default, this pragma may become a no-op. See [utf8](#).

Unless explicitly stated, Perl operators use character semantics for Unicode data and byte semantics for non-Unicode data. The decision to use character semantics is made transparently. If input data comes from

a Unicode source—for example, if a character encoding layer is added to a filehandle or a literal Unicode string constant appears in a program—character semantics apply. Otherwise, byte semantics are in effect. The `bytes` pragma should be used to force byte semantics on Unicode data.

If strings operating under byte semantics and strings with Unicode character data are concatenated, the new string will be upgraded to *ISO 8859-1 (Latin-1)*, even if the old Unicode string used EBCDIC. This translation is done without regard to the system's native 8-bit encoding, so to change this for systems with non-Latin-1 and non-EBCDIC native encodings use the `encoding` pragma. See [encoding](#).

Under character semantics, many operations that formerly operated on bytes now operate on characters. A character in Perl is logically just a number ranging from 0 to 2^{31} or so. Larger characters may encode into longer sequences of bytes internally, but this internal detail is mostly hidden for Perl code. See [perluniintro](#) for more.

Effects of Character Semantics

Character semantics have the following effects:

- Strings—including hash keys—and regular expression patterns may contain characters that have an ordinal value larger than 255.

If you use a Unicode editor to edit your program, Unicode characters may occur directly within the literal strings in one of the various Unicode encodings (UTF-8, UTF-EBCDIC, UCS-2, etc.), but will be recognized as such and converted to Perl's internal representation only if the appropriate [encoding](#) is specified.

Unicode characters can also be added to a string by using the `\x{ . . . }` notation. The Unicode code for the desired character, in hexadecimal, should be placed in the braces. For instance, a smiley face is `\x{263A}`. This encoding scheme only works for characters with a code of 0x100 or above.

Additionally, if you

```
use charnames ':full';
```

you can use the `\N{ . . . }` notation and put the official Unicode character name within the braces, such as `\N{WHITE SMILING FACE}`.

- If an appropriate [encoding](#) is specified, identifiers within the Perl script may contain Unicode alphanumeric characters, including ideographs. Perl does not currently attempt to canonicalize variable names.
- Regular expressions match characters instead of bytes. `"."` matches a character instead of a byte. The `\C` pattern is provided to force a match a single byte—a `char` in C, hence `\C`.
- Character classes in regular expressions match characters instead of bytes and match against the character properties specified in the Unicode properties database. `\w` can be used to match a Japanese ideograph, for instance.
- Named Unicode properties, scripts, and block ranges may be used like character classes via the `\p{ }` "matches property" construct and the `\P{ }` negation, "doesn't match property".

For instance, `\p{Lu}` matches any character with the Unicode "Lu" (Letter, uppercase) property, while `\p{M}` matches any character with an "M" (mark—accents and such) property. Brackets are not required for single letter properties, so `\p{M}` is equivalent to `\pM`. Many predefined properties are available, such as `\p{Mirrored}` and `\p{Tibetan}`.

The official Unicode script and block names have spaces and dashes as separators, but for convenience you can use dashes, spaces, or underbars, and case is unimportant. It is recommended, however, that for consistency you use the following naming: the official Unicode script, property, or block name (see below for the additional rules that apply to block names) with whitespace and dashes removed, and the words "uppercase-first-lowercase-rest". `Latin-1 Supplement` thus becomes `Latin1Supplement`.

You can also use negation in both `\p{ }` and `\P{ }` by introducing a caret (^) between the first brace and the property name: `\p{^Tamil}` is equal to `\P{Tamil}`.

Here are the basic Unicode General Category properties, followed by their long form. You can use either; `\p{Lu}` and `\p{LowercaseLetter}`, for instance, are identical.

Short	Long
L	Letter
Lu	UppercaseLetter
Ll	LowercaseLetter
Lt	TitlecaseLetter
Lm	ModifierLetter
Lo	OtherLetter
M	Mark
Mn	NonspacingMark
Mc	SpacingMark
Me	EnclosingMark
N	Number
Nd	DecimalNumber
Nl	LetterNumber
No	OtherNumber
P	Punctuation
Pc	ConnectorPunctuation
Pd	DashPunctuation
Ps	OpenPunctuation
Pe	ClosePunctuation
Pi	InitialPunctuation
	(may behave like Ps or Pe depending on usage)
Pf	FinalPunctuation
	(may behave like Ps or Pe depending on usage)
Po	OtherPunctuation
S	Symbol
Sm	MathSymbol
Sc	CurrencySymbol
Sk	ModifierSymbol
So	OtherSymbol
Z	Separator
Zs	SpaceSeparator
Zl	LineSeparator
Zp	ParagraphSeparator
C	Other
Cc	Control
Cf	Format
Cs	Surrogate (not usable)
Co	PrivateUse
Cn	Unassigned

Single-letter properties match all characters in any of the two-letter sub-properties starting with the same letter. `L&` is a special case, which is an alias for `Ll`, `Lu`, and `Lt`.

Because Perl hides the need for the user to understand the internal representation of Unicode characters, there is no need to implement the somewhat messy concept of surrogates. `Cs` is therefore not supported.

Because scripts differ in their directionality—Hebrew is written right to left, for example—Unicode supplies these properties:

Property	Meaning
BidiL	Left-to-Right
BidiLRE	Left-to-Right Embedding
BidiLRO	Left-to-Right Override
BidiR	Right-to-Left
BidiAL	Right-to-Left Arabic
BidiRLE	Right-to-Left Embedding
BidiRLO	Right-to-Left Override
BidiPDF	Pop Directional Format
BidiEN	European Number
BidiES	European Number Separator
BidiET	European Number Terminator
BidiAN	Arabic Number
BidiCS	Common Number Separator
BidiNSM	Non-Spacing Mark
BidiBN	Boundary Neutral
BidiB	Paragraph Separator
BidiS	Segment Separator
BidiWS	Whitespace
BidiON	Other Neutrals

For example, `\p{BidiR}` matches characters that are normally written right to left.

Scripts

The script names which can be used by `\p{...}` and `\P{...}`, such as in `\p{Latin}` or `\p{Cyrillic}`, are as follows:

```
Arabic
Armenian
Bengali
Bopomofo
Buhid
CanadianAboriginal
Cherokee
Cyrillic
Deseret
Devanagari
Ethiopic
Georgian
Gothic
Greek
Gujarati
Gurmukhi
Han
Hangul
Hanunoo
Hebrew
Hiragana
Inherited
Kannada
Katakana
Khmer
Lao
```

Latin
Malayalam
Mongolian
Myanmar
Ogham
OldItalic
Oriya
Runic
Sinhala
Syriac
Tagalog
Tagbanwa
Tamil
Telugu
Thaana
Thai
Tibetan
Yi

Extended property classes can supplement the basic properties, defined by the *PropList* Unicode database:

ASCIISHexDigit
BidiControl
Dash
Deprecated
Diacritic
Extender
GraphemeLink
HexDigit
Hyphen
Ideographic
IDSBinaryOperator
IDSTrinaryOperator
JoinControl
LogicalOrderException
NoncharacterCodePoint
OtherAlphabetic
OtherDefaultIgnorableCodePoint
OtherGraphemeExtend
OtherLowercase
OtherMath
OtherUppercase
QuotationMark
Radical
SoftDotted
TerminalPunctuation
UnifiedIdeograph
WhiteSpace

and there are further derived properties:

Alphabetic	Lu + Ll + Lt + Lm + Lo + OtherAlphabetic
Lowercase	Ll + OtherLowercase
Uppercase	Lu + OtherUppercase
Math	Sm + OtherMath

ID_Start	Lu + Ll + Lt + Lm + Lo + Nl
ID_Continue	ID_Start + Mn + Mc + Nd + Pc
Any	Any character
Assigned	Any non-Cn character (i.e. synonym for \P{Cn})
Unassigned	Synonym for \p{Cn}
Common	Any character (or unassigned code point) not explicitly assigned to a script

For backward compatibility (with Perl 5.6), all properties mentioned so far may have `Is` prepended to their name, so `\P{IsLu}`, for example, is equal to `\P{Lu}`.

Blocks

In addition to **scripts**, Unicode also defines **blocks** of characters. The difference between scripts and blocks is that the concept of scripts is closer to natural languages, while the concept of blocks is more of an artificial grouping based on groups of 256 Unicode characters. For example, the `Latin` script contains letters from many blocks but does not contain all the characters from those blocks. It does not, for example, contain digits, because digits are shared across many scripts. Digits and similar groups, like punctuation, are in a category called `Common`.

For more about scripts, see the UTR #24:

<http://www.unicode.org/unicode/reports/tr24/>

For more about blocks, see:

<http://www.unicode.org/Public/UNIDATA/Blocks.txt>

Block names are given with the `In` prefix. For example, the `Katakana` block is referenced via `\p{InKatakana}`. The `In` prefix may be omitted if there is no naming conflict with a script or any other property, but it is recommended that `In` always be used for block tests to avoid confusion.

These block names are supported:

```
InAlphabeticPresentationForms
InArabic
InArabicPresentationFormsA
InArabicPresentationFormsB
InArmenian
InArrows
InBasicLatin
InBengali
InBlockElements
InBopomofo
InBopomofoExtended
InBoxDrawing
InBraillePatterns
InBuhid
InByzantineMusicalSymbols
InCJKCompatibility
InCJKCompatibilityForms
InCJKCompatibilityIdeographs
InCJKCompatibilityIdeographsSupplement
InCJKRadicalsSupplement
InCJKSymbolsAndPunctuation
InCJKUnifiedIdeographs
InCJKUnifiedIdeographsExtensionA
InCJKUnifiedIdeographsExtensionB
InCherokee
```

InCombiningDiacriticalMarks
InCombiningDiacriticalMarksforSymbols
InCombiningHalfMarks
InControlPictures
InCurrencySymbols
InCyrillic
InCyrillicSupplementary
InDeseret
InDevanagari
InDingbats
InEnclosedAlphanumerics
InEnclosedCJKLettersAndMonths
InEthiopic
InGeneralPunctuation
InGeometricShapes
InGeorgian
InGothic
InGreekExtended
InGreekAndCoptic
InGujarati
InGurmukhi
InHalfwidthAndFullwidthForms
InHangulCompatibilityJamo
InHangulJamo
InHangulSyllables
InHanunoo
InHebrew
InHighPrivateUseSurrogates
InHighSurrogates
InHiragana
InIPAExtensions
InIdeographicDescriptionCharacters
InKanbun
InKangxiRadicals
InKannada
InKatakana
InKatakanaPhoneticExtensions
InKhmer
InLao
InLatin1Supplement
InLatinExtendedA
InLatinExtendedAdditional
InLatinExtendedB
InLetterlikeSymbols
InLowSurrogates
InMalayalam
InMathematicalAlphanumericSymbols
InMathematicalOperators
InMiscellaneousMathematicalSymbolsA
InMiscellaneousMathematicalSymbolsB
InMiscellaneousSymbols
InMiscellaneousTechnical
InMongolian
InMusicalSymbols

InMyanmar
 InNumberForms
 InOgham
 InOldItalic
 InOpticalCharacterRecognition
 InOriya
 InPrivateUseArea
 InRunic
 InSinhala
 InSmallFormVariants
 InSpacingModifierLetters
 InSpecials
 InSuperscriptsAndSubscripts
 InSupplementalArrowsA
 InSupplementalArrowsB
 InSupplementalMathematicalOperators
 InSupplementaryPrivateUseAreaA
 InSupplementaryPrivateUseAreaB
 InSyriac
 InTagalog
 InTagbanwa
 InTags
 InTamil
 InTelugu
 InThaana
 InThai
 InTibetan
 InUnifiedCanadianAboriginalSyllabics
 InVariationSelectors
 InYiRadicals
 InYiSyllables

- The special pattern `\X` matches any extended Unicode sequence—"a combining character sequence" in Standardese—where the first character is a base character and subsequent characters are mark characters that apply to the base character. `\X` is equivalent to `(?:\PM\pM*)`.
- The `tr///` operator translates characters instead of bytes. Note that the `tr///CU` functionality has been removed. For similar functionality see `pack('U0', ...)` and `pack('C0', ...)`.
- Case translation operators use the Unicode case translation tables when character input is provided. Note that `uc()`, or `\U` in interpolated strings, translates to uppercase, while `ucfirst`, or `\u` in interpolated strings, translates to titlecase in languages that make the distinction.
- Most operators that deal with positions or lengths in a string will automatically switch to using character positions, including `chop()`, `substr()`, `pos()`, `index()`, `rindex()`, `sprintf()`, `write()`, and `length()`. Operators that specifically do not switch include `vec()`, `pack()`, and `unpack()`. Operators that really don't care include `chomp()`, operators that treat strings as a bucket of bits such as `sort()`, and operators dealing with filenames.
- The `pack()/unpack()` letters `c` and `C` do *not* change, since they are often used for byte-oriented formats. Again, think `char` in the C language.

There is a new `U` specifier that converts between Unicode characters and code points.

- The `chr()` and `ord()` functions work on characters, similar to `pack("U")` and `unpack("U")`, *not* `pack("C")` and `unpack("C")`. `pack("C")` and `unpack("C")` are methods for emulating byte-oriented `chr()` and `ord()` on Unicode strings. While these methods reveal the internal encoding of Unicode strings, that is not something one normally needs to care about at all.

- The bit string operators, `&`, `|`, `^`, `~`, can operate on character data. However, for backward compatibility, such as when using bit string operations when characters are all less than 256 in ordinal value, one should not use `~` (the bit complement) with characters of both values less than 256 and values greater than 256. Most importantly, DeMorgan's laws ($\sim(\$x|\$y) \text{ eq } \sim\$x\&\sim\y and $\sim(\$x\&\$y) \text{ eq } \sim\$x|\sim\y) will not hold. The reason for this mathematical *faux pas* is that the complement cannot return **both** the 8-bit (byte-wide) bit complement **and** the full character-wide bit complement.
- `lc()`, `uc()`, `lcfirst()`, and `ucfirst()` work for the following cases:
 - the case mapping is from a single Unicode character to another single Unicode character, or
 - the case mapping is from a single Unicode character to more than one Unicode character.
 The following cases do not yet work:
 - the "final sigma" (Greek), and
 - anything to with locales (Lithuanian, Turkish, Azeri).
 See the Unicode Technical Report #21, Case Mappings, for more details.
- And finally, `scalar reverse()` reverses by character rather than by byte.

User-Defined Character Properties

You can define your own character properties by defining subroutines whose names begin with "In" or "Is". The subroutines must be visible in the package that uses the properties. The user-defined properties can be used in the regular expression `\p` and `\P` constructs.

The subroutines must return a specially-formatted string, with one or more newline-separated lines. Each line must be one of the following:

- Two hexadecimal numbers separated by horizontal whitespace (space or tabular characters) denoting a range of Unicode code points to include.
- Something to include, prefixed by "+": a built-in character property (prefixed by "utf8::"), to represent all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to exclude, prefixed by "-": an existing character property (prefixed by "utf8::"), for all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to negate, prefixed by "!": an existing character property (prefixed by "utf8::") for all the characters except the characters in the property; two hexadecimal code points for a range; or a single hexadecimal code point.

For example, to define a property that covers both the Japanese syllabaries (hiragana and katakana), you can define

```
sub InKana {
    return <<END;
    3040\t309F
    30A0\t30FF
    END
}
```

Imagine that the here-doc end marker is at the beginning of the line. Now you can use `\p{InKana}` and `\P{InKana}`.

You could also have used the existing block property names:

```

sub InKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
END
}

```

Suppose you wanted to match only the allocated characters, not the raw block ranges: in other words, you want to remove the non-characters:

```

sub InKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
-utf8::IsCn
END
}

```

The negation is useful for defining (surprise!) negated classes.

```

sub InNotKana {
    return <<'END';
!utf8::InHiragana
-utf8::InKatakana
+utf8::IsCn
END
}

```

Character Encodings for Input and Output

See [Encode](#).

Unicode Regular Expression Support Level

The following list of Unicode support for regular expressions describes all the features currently supported. The references to "Level N" and the section numbers refer to the Unicode Technical Report 18, "Unicode Regular Expression Guidelines".

- Level 1 – Basic Unicode Support

2.1 Hex Notation	- done	[1]
Named Notation	- done	[2]
2.2 Categories	- done	[3] [4]
2.3 Subtraction	- MISSING	[5] [6]
2.4 Simple Word Boundaries	- done	[7]
2.5 Simple Loose Matches	- done	[8]
2.6 End of Line	- MISSING	[9] [10]
[1] \x{...}		
[2] \N{...}		
[3] . \p{...} \P{...}		
[4] now scripts (see UTR#24 Script Names) in addition to blocks		
[5] have negation		
[6] can use regular expression look-ahead [a] or user-defined character properties [b] to emulate subtraction		
[7] include Letters in word characters		
[8] note that Perl does Full case-folding in matching, not Simple: for example U+1F88 is equivalent with U+1F000 U+03B9, not with 1F80. This difference matters for certain Greek capital letters with certain modifiers: the Full case-folding decomposes the letter, while the Simple case-folding would map		

it to a single character.

- [9] see UTR#13 Unicode Newline Guidelines
- [10] should do ^ and \$ also on \x{85}, \x{2028} and \x{2029})
(should also affect <>, \$., and script line numbers)
(the \x{85}, \x{2028} and \x{2029} do match \s)

[a] You can mimic class subtraction using lookahead. For example, what TR18 might write as

```
[{Greek}-[{UNASSIGNED}]]
```

in Perl can be written as:

```
(?!\p{Unassigned})\p{InGreekAndCoptic}  
(?=\p{Assigned})\p{InGreekAndCoptic}
```

But in this particular example, you probably really want

```
\p{GreekAndCoptic}
```

which will match assigned characters known to be part of the Greek script.

[b] See .

- Level 2 – Extended Unicode Support

3.1	Surrogates	- MISSING	
3.2	Canonical Equivalents	- MISSING	[11] [12]
3.3	Locale-Independent Graphemes	- MISSING	[13]
3.4	Locale-Independent Words	- MISSING	[14]
3.5	Locale-Independent Loose Matches	- MISSING	[15]

[11] see UTR#15 Unicode Normalization

[12] have Unicode::Normalize but not integrated to regexes

[13] have \X but at this level . should equal that

[14] need three classes, not just \w and \W

[15] see UTR#21 Case Mappings

- Level 3 – Locale-Sensitive Support

4.1	Locale-Dependent Categories	- MISSING	
4.2	Locale-Dependent Graphemes	- MISSING	[16] [17]
4.3	Locale-Dependent Words	- MISSING	
4.4	Locale-Dependent Loose Matches	- MISSING	
4.5	Locale-Dependent Ranges	- MISSING	

[16] see UTR#10 Unicode Collation Algorithms

[17] have Unicode::Collate but not integrated to regexes

Unicode Encodings

Unicode characters are assigned to *code points*, which are abstract numbers. To use these numbers, various encodings are needed.

- UTF-8

UTF-8 is a variable-length (1 to 6 bytes, current character allocations require 4 bytes), byte-order independent encoding. For ASCII (and we really do mean 7-bit ASCII, not another 8-bit encoding), UTF-8 is transparent.

The following table is from Unicode 3.2.

Code Points	1st Byte	2nd Byte	3rd Byte	4th Byte
U+0000..U+007F	00..7F			
U+0080..U+07FF	C2..DF	80..BF		

U+0800..U+0FFF	E0	A0..BF	80..BF	
U+1000..U+CFFF	E1..EC	80..BF	80..BF	
U+D000..U+D7FF	ED	80..9F	80..BF	
U+D800..U+DFFF	***** ill-formed *****			
U+E000..U+FFFF	EE..EF	80..BF	80..BF	
U+10000..U+3FFFF	F0	90..BF	80..BF	80..BF
U+40000..U+FFFFF	F1..F3	80..BF	80..BF	80..BF
U+100000..U+10FFFF	F4	80..8F	80..BF	80..BF

Note the A0..BF in U+0800..U+0FFF, the 80..9F in U+D000..U+D7FF, the 90..BF in U+10000..U+3FFFF, and the 80...8F in U+100000..U+10FFFF. The "gaps" are caused by legal UTF-8 avoiding non-shortest encodings: it is technically possible to UTF-8-encode a single code point in different ways, but that is explicitly forbidden, and the shortest possible encoding should always be used. So that's what Perl does.

Another way to look at it is via bits:

Code Points	1st Byte	2nd Byte	3rd Byte	4th Byte
0aaaaaaaa	0aaaaaaaa			
00000bbbbbaaaaaa	110bbbbbb	10aaaaaaa		
ccccbbbbbbaaaaaa	1110cccc	10bbbbbbb	10aaaaaaa	
00000dddcccccbbbbbaaaaaa	11110ddd	10ccccccc	10bbbbbbb	10aaaaaaa

As you can see, the continuation bytes all begin with 10, and the leading bits of the start byte tell how many bytes they are in the encoded character.

- UTF-EBCDIC

Like UTF-8 but EBCDIC-safe, in the way that UTF-8 is ASCII-safe.

- UTF-16, UTF-16BE, UTF16-LE, Surrogates, and BOMs (Byte Order Marks)

The followings items are mostly for reference and general Unicode knowledge, Perl doesn't use these constructs internally.

UTF-16 is a 2 or 4 byte encoding. The Unicode code points U+0000..U+FFFF are stored in a single 16-bit unit, and the code points U+10000..U+10FFFF in two 16-bit units. The latter case is using *surrogates*, the first 16-bit unit being the *high surrogate*, and the second being the *low surrogate*.

Surrogates are code points set aside to encode the U+10000..U+10FFFF range of Unicode code points in pairs of 16-bit units. The *high surrogates* are the range U+D800..U+DBFF, and the *low surrogates* are the range U+DC00..U+DFFF. The surrogate encoding is

```
$hi = ($uni - 0x10000) / 0x400 + 0xD800;
$lo = ($uni - 0x10000) % 0x400 + 0xDC00;
```

and the decoding is

```
$uni = 0x10000 + ($hi - 0xD800) * 0x400 + ($lo - 0xDC00);
```

If you try to generate surrogates (for example by using `chr()`), you will get a warning if warnings are turned on, because those code points are not valid for a Unicode character.

Because of the 16-bitness, UTF-16 is byte-order dependent. UTF-16 itself can be used for in-memory computations, but if storage or transfer is required either UTF-16BE (big-endian) or UTF-16LE (little-endian) encodings must be chosen.

This introduces another problem: what if you just know that your data is UTF-16, but you don't know which endianness? Byte Order Marks, or BOMs, are a solution to this. A special character has been reserved in Unicode to function as a byte order marker: the character with the code point U+FEFF is the BOM.

The trick is that if you read a BOM, you will know the byte order, since if it was written on a big-endian platform, you will read the bytes 0xFE 0xFF, but if it was written on a little-endian platform, you will read the bytes 0xFF 0xFE. (And if the originating platform was writing in UTF-8, you will read the bytes 0xEF 0xBB 0xBF.)

The way this trick works is that the character with the code point U+FFFE is guaranteed not to be a valid Unicode character, so the sequence of bytes 0xFF 0xFE is unambiguously "BOM, represented in little-endian format" and cannot be U+FFFE, represented in big-endian format".

- UTF-32, UTF-32BE, UTF32-LE

The UTF-32 family is pretty much like the UTF-16 family, expect that the units are 32-bit, and therefore the surrogate scheme is not needed. The BOM signatures will be 0x00 0x00 0xFE 0xFF for BE and 0xFF 0xFE 0x00 0x00 for LE.

- UCS-2, UCS-4

Encodings defined by the ISO 10646 standard. UCS-2 is a 16-bit encoding. Unlike UTF-16, UCS-2 is not extensible beyond U+FFFF, because it does not use surrogates. UCS-4 is a 32-bit encoding, functionally identical to UTF-32.

- UTF-7

A seven-bit safe (non-eight-bit) encoding, which is useful if the transport or storage is not eight-bit safe. Defined by RFC 2152.

Security Implications of Unicode

- Malformed UTF-8

Unfortunately, the specification of UTF-8 leaves some room for interpretation of how many bytes of encoded output one should generate from one input Unicode character. Strictly speaking, the shortest possible sequence of UTF-8 bytes should be generated, because otherwise there is potential for an input buffer overflow at the receiving end of a UTF-8 connection. Perl always generates the shortest length UTF-8, and with warnings on Perl will warn about non-shortest length UTF-8 along with other malformations, such as the surrogates, which are not real Unicode code points.

- Regular expressions behave slightly differently between byte data and character (Unicode) data. For example, the "word character" character class `\w` will work differently depending on if data is eight-bit bytes or Unicode.

In the first case, the set of `\w` characters is either small—the default set of alphabetic characters, digits, and the `"_"`—or, if you are using a locale (see [perllocale](#)), the `\w` might contain a few more letters according to your language and country.

In the second case, the `\w` set of characters is much, much larger. Most importantly, even in the set of the first 256 characters, it will probably match different characters: unlike most locales, which are specific to a language and country pair, Unicode classifies all the characters that are letters *somewhere* as `\w`. For example, your locale might not think that LATIN SMALL LETTER ETH is a letter (unless you happen to speak Icelandic), but Unicode does.

As discussed elsewhere, Perl has one foot (two hooves?) planted in each of two worlds: the old world of bytes and the new world of characters, upgrading from bytes to characters when necessary. If your legacy code does not explicitly use Unicode, no automatic switch-over to characters should happen. Characters shouldn't get downgraded to bytes, either. It is possible to accidentally mix bytes and characters, however (see [perluniintro](#)), in which case `\w` in regular expressions might start behaving differently. Review your code. Use warnings and the `strict` pragma.

Unicode in Perl on EBCDIC

The way Unicode is handled on EBCDIC platforms is still experimental. On such platforms, references to UTF-8 encoding in this document and elsewhere should be read as meaning the UTF-EBCDIC specified in Unicode Technical Report 16, unless ASCII vs. EBCDIC issues are specifically discussed. There is no

`utf8bc` pragma or `":utf8bc"` layer; rather, `"utf8"` and `":utf8"` are reused to mean the platform's "natural" 8-bit encoding of Unicode. See [perlebcdic](#) for more discussion of the issues.

Locales

Usually locale settings and Unicode do not affect each other, but there are a couple of exceptions:

- If your locale environment variables (`LANGUAGE`, `LC_ALL`, `LC_CTYPE`, `LANG`) contain the strings `'UTF-8'` or `'UTF8'` (case-insensitive matching), the default encodings of your `STDIN`, `STDOUT`, and `STDERR`, and of **any subsequent file open**, are considered to be UTF-8.
- Perl tries really hard to work both with Unicode and the old byte-oriented world. Most often this is nice, but sometimes Perl's straddling of the proverbial fence causes problems.

Using Unicode in XS

If you want to handle Perl Unicode in XS extensions, you may find the following C APIs useful. See [perlapi](#) for details.

- `DO_UTF8(sv)` returns true if the UTF8 flag is on and the bytes pragma is not in effect. `SvUTF8(sv)` returns true if the UTF8 flag is on; the bytes pragma is ignored. The UTF8 flag being on does **not** mean that there are any characters of code points greater than 255 (or 127) in the scalar or that there are even any characters in the scalar. What the UTF8 flag means is that the sequence of octets in the representation of the scalar is the sequence of UTF-8 encoded code points of the characters of a string. The UTF8 flag being off means that each octet in this representation encodes a single character with code point 0..255 within the string. Perl's Unicode model is not to use UTF-8 until it is absolutely necessary.
- `uvuni_to_utf8(buf, chr)` writes a Unicode character code point into a buffer encoding the code point as UTF-8, and returns a pointer pointing after the UTF-8 bytes.
- `utf8_to_uvuni(buf, lenp)` reads UTF-8 encoded bytes from a buffer and returns the Unicode character code point and, optionally, the length of the UTF-8 byte sequence.
- `utf8_length(start, end)` returns the length of the UTF-8 encoded buffer in characters. `sv_len_utf8(sv)` returns the length of the UTF-8 encoded scalar.
- `sv_utf8_upgrade(sv)` converts the string of the scalar to its UTF-8 encoded form. `sv_utf8_downgrade(sv)` does the opposite, if possible. `sv_utf8_encode(sv)` is like `sv_utf8_upgrade` except that it does not set the UTF8 flag. `sv_utf8_decode()` does the opposite of `sv_utf8_encode()`. Note that none of these are to be used as general-purpose encoding or decoding interfaces: use `Encode` for that. `sv_utf8_upgrade()` is affected by the encoding pragma but `sv_utf8_downgrade()` is not (since the encoding pragma is designed to be a one-way street).
- `is_utf8_char(s)` returns true if the pointer points to a valid UTF-8 character.
- `is_utf8_string(buf, len)` returns true if `len` bytes of the buffer are valid UTF-8.
- `UTF8SKIP(buf)` will return the number of bytes in the UTF-8 encoded character in the buffer. `UNISKIP(chr)` will return the number of bytes required to UTF-8-encode the Unicode character code point. `UTF8SKIP()` is useful for example for iterating over the characters of a UTF-8 encoded buffer; `UNISKIP()` is useful, for example, in computing the size required for a UTF-8 encoded buffer.
- `utf8_distance(a, b)` will tell the distance in characters between the two pointers pointing to the same UTF-8 encoded buffer.
- `utf8_hop(s, off)` will return a pointer to an UTF-8 encoded buffer that is `off` (positive or negative) Unicode characters displaced from the UTF-8 buffer `s`. Be careful not to overstep the buffer: `utf8_hop()` will merrily run off the end or the beginning of the buffer if told to do so.

- `pv_uni_display(dsv, spv, len, pvlm, flags)` and `sv_uni_display(dsv, ssv, pvlm, flags)` are useful for debugging the output of Unicode strings and scalars. By default they are useful only for debugging—they display **all** characters as hexadecimal code points—but with the flags `UNI_DISPLAY_ISPRINT`, `UNI_DISPLAY_BACKSLASH`, and `UNI_DISPLAY_QQ` you can make the output more readable.
- `ibcmp_utf8(s1, pe1, u1, l1, u1, s2, pe2, l2, u2)` can be used to compare two strings case-insensitively in Unicode. For case-sensitive comparisons you can just use `memEQ()` and `memNE()` as usual.

For more information, see [perlapi](#), and [utf8.c](#) and [utf8.h](#) in the Perl source code distribution.

BUGS

Interaction with Locales

Use of locales with Unicode data may lead to odd results. Currently, Perl attempts to attach 8-bit locale info to characters in the range 0..255, but this technique is demonstrably incorrect for locales that use characters above that range when mapped into Unicode. Perl's Unicode support will also tend to run slower. Use of locales with Unicode is discouraged.

Interaction with Extensions

When Perl exchanges data with an extension, the extension should be able to understand the UTF-8 flag and act accordingly. If the extension doesn't know about the flag, it's likely that the extension will return incorrectly-flagged data.

So if you're working with Unicode data, consult the documentation of every module you're using if there are any issues with Unicode data exchange. If the documentation does not talk about Unicode at all, suspect the worst and probably look at the source to learn how the module is implemented. Modules written completely in Perl shouldn't cause problems. Modules that directly or indirectly access code written in other programming languages are at risk.

For affected functions, the simple strategy to avoid data corruption is to always make the encoding of the exchanged data explicit. Choose an encoding that you know the extension can handle. Convert arguments passed to the extensions to that encoding and convert results back from that encoding. Write wrapper functions that do the conversions for you, so you can later change the functions when the extension catches up.

To provide an example, let's say the popular `Foo::Bar::escape_html` function doesn't deal with Unicode data yet. The wrapper function would convert the argument to raw UTF-8 and convert the result back to Perl's internal representation like so:

```
sub my_escape_html ($) {
    my($what) = shift;
    return unless defined $what;
    Encode::decode_utf8(Foo::Bar::escape_html(Encode::encode_utf8($what)));
}
```

Sometimes, when the extension does not convert data but just stores and retrieves them, you will be in a position to use the otherwise dangerous `Encode::_utf8_on()` function. Let's say the popular `Foo::Bar` extension, written in C, provides a `param` method that lets you store and retrieve data according to these prototypes:

```
$self->param($name, $value);          # set a scalar
$value = $self->param($name);         # retrieve a scalar
```

If it does not yet provide support for any encoding, one could write a derived class with such a `param` method:

```
sub param {
    my($self, $name, $value) = @_;
```

```

utf8::upgrade($name);      # make sure it is UTF-8 encoded
if (defined $value)
    utf8::upgrade($value); # make sure it is UTF-8 encoded
    return $self->SUPER::param($name,$value);
} else {
    my $ret = $self->SUPER::param($name);
    Encode::_utf8_on($ret); # we know, it is UTF-8 encoded
    return $ret;
}
}

```

Some extensions provide filters on data entry/exit points, such as `DB_File::filter_store_key` and `family`. Look out for such filters in the documentation of your extensions, they can make the transition to Unicode data much easier.

Speed

Some functions are slower when working on UTF-8 encoded strings than on byte encoded strings. All functions that need to hop over characters such as `length()`, `substr()` or `index()` can work **much** faster when the underlying data are byte-encoded. Witness the following benchmark:

```

% perl -e '
use Benchmark;
use strict;
our $l = 10000;
our $u = our $b = "x" x $l;
substr($u,0,1) = "\x{100}";
timethese(-2,{
    LENGTH_B => q{ length($b) },
    LENGTH_U => q{ length($u) },
    SUBSTR_B => q{ substr($b, $l/4, $l/2) },
    SUBSTR_U => q{ substr($u, $l/4, $l/2) },
});
'

```

```

Benchmark: running LENGTH_B, LENGTH_U, SUBSTR_B, SUBSTR_U for at least 2 CPU second
    LENGTH_B:  2 wallclock secs ( 2.36 usr +  0.00 sys =  2.36 CPU) @ 5649983.05/s (n=
    LENGTH_U:  2 wallclock secs ( 2.11 usr +  0.00 sys =  2.11 CPU) @ 12155.45/s (n=2
    SUBSTR_B:  3 wallclock secs ( 2.16 usr +  0.00 sys =  2.16 CPU) @ 374480.09/s (n=
    SUBSTR_U:  2 wallclock secs ( 2.11 usr +  0.00 sys =  2.11 CPU) @ 6791.00/s (n=14

```

The numbers show an incredible slowness on long UTF-8 strings. You should carefully avoid using these functions in tight loops. If you want to iterate over characters, the superior coding technique would split the characters into an array instead of using `substr`, as the following benchmark shows:

```

% perl -e '
use Benchmark;
use strict;
our $l = 10000;
our $u = our $b = "x" x $l;
substr($u,0,1) = "\x{100}";
timethese(-5,{
    SPLIT_B => q{ for my $c (split //, $b){} },
    SPLIT_U => q{ for my $c (split //, $u){} },
    SUBSTR_B => q{ for my $i (0..length($b)-1){my $c = substr($b,$i,1);} },
    SUBSTR_U => q{ for my $i (0..length($u)-1){my $c = substr($u,$i,1);} },
});
'

```

```

Benchmark: running SPLIT_B, SPLIT_U, SUBSTR_B, SUBSTR_U for at least 5 CPU seconds.

```

```
SPLIT_B:  6 wallclock secs ( 5.29 usr +  0.00 sys =  5.29 CPU) @ 56.14/s (n=297)
SPLIT_U:  5 wallclock secs ( 5.17 usr +  0.01 sys =  5.18 CPU) @ 55.21/s (n=286)
SUBSTR_B:  5 wallclock secs ( 5.34 usr +  0.00 sys =  5.34 CPU) @ 123.22/s (n=658)
SUBSTR_U:  7 wallclock secs ( 6.20 usr +  0.00 sys =  6.20 CPU) @  0.81/s (n=5)
```

Even though the algorithm based on `substr()` is faster than `split()` for byte-encoded data, it pales in comparison to the speed of `split()` when used with UTF-8 data.

SEE ALSO

perluniintro, *encoding*, *Encode*, *open*, *utf8*, *bytes*, *perlretut*, `${^WIDE_SYSTEM_CALLS}` in *perlvar*

NAME

perluniintro – Perl Unicode introduction

DESCRIPTION

This document gives a general idea of Unicode and how to use Unicode in Perl.

Unicode

Unicode is a character set standard which plans to codify all of the writing systems of the world, plus many other symbols.

Unicode and ISO/IEC 10646 are coordinated standards that provide code points for characters in almost all modern character set standards, covering more than 30 writing systems and hundreds of languages, including all commercially–important modern languages. All characters in the largest Chinese, Japanese, and Korean dictionaries are also encoded. The standards will eventually cover almost all characters in more than 250 writing systems and thousands of languages.

A Unicode *character* is an abstract entity. It is not bound to any particular integer width, especially not to the C language `char`. Unicode is language–neutral and display–neutral: it does not encode the language of the text and it does not define fonts or other graphical layout details. Unicode operates on characters and on text built from those characters.

Unicode defines characters like LATIN CAPITAL LETTER A or GREEK SMALL LETTER ALPHA and unique numbers for the characters, in this case 0x0041 and 0x03B1, respectively. These unique numbers are called *code points*.

The Unicode standard prefers using hexadecimal notation for the code points. If numbers like 0x0041 are unfamiliar to you, take a peek at a later section, . The Unicode standard uses the notation U+0041 LATIN CAPITAL LETTER A, to give the hexadecimal code point and the normative name of the character.

Unicode also defines various *properties* for the characters, like "uppercase" or "lowercase", "decimal digit", or "punctuation"; these properties are independent of the names of the characters. Furthermore, various operations on the characters like uppercasing, lowercasing, and collating (sorting) are defined.

A Unicode character consists either of a single code point, or a *base character* (like LATIN CAPITAL LETTER A), followed by one or more *modifiers* (like COMBINING ACUTE ACCENT). This sequence of base character and modifiers is called a *combining character sequence*.

Whether to call these combining character sequences "characters" depends on your point of view. If you are a programmer, you probably would tend towards seeing each element in the sequences as one unit, or "character". The whole sequence could be seen as one "character", however, from the user's point of view, since that's probably what it looks like in the context of the user's language.

With this "whole sequence" view of characters, the total number of characters is open–ended. But in the programmer's "one unit is one character" point of view, the concept of "characters" is more deterministic. In this document, we take that second point of view: one "character" is one Unicode code point, be it a base character or a combining character.

For some combinations, there are *precomposed* characters. LATIN CAPITAL LETTER A WITH ACUTE, for example, is defined as a single code point. These precomposed characters are, however, only available for some combinations, and are mainly meant to support round–trip conversions between Unicode and legacy standards (like the ISO 8859). In the general case, the composing method is more extensible. To support conversion between different compositions of the characters, various *normalization forms* to standardize representations are also defined.

Because of backward compatibility with legacy encodings, the "a unique number for every character" idea breaks down a bit: instead, there is "at least one number for every character". The same character could be represented differently in several legacy encodings. The converse is also not true: some code points do not have an assigned character. Firstly, there are unallocated code points within otherwise used blocks. Secondly, there are special Unicode control characters that do not represent true characters.

A common myth about Unicode is that it would be "16-bit", that is, Unicode is only represented as 0x10000 (or 65536) characters from 0x0000 to 0xFFFF. **This is untrue.** Since Unicode 2.0, Unicode has been defined all the way up to 21 bits (0x10FFFF), and since Unicode 3.1, characters have been defined beyond 0xFFFF. The first 0x10000 characters are called the *Plane 0*, or the *Basic Multilingual Plane* (BMP). With Unicode 3.1, 17 planes in all are defined—but nowhere near full of defined characters, yet.

Another myth is that the 256-character blocks have something to do with languages—that each block would define the characters used by a language or a set of languages. **This is also untrue.** The division into blocks exists, but it is almost completely accidental—an artifact of how the characters have been and still are allocated. Instead, there is a concept called *scripts*, which is more useful: there is Latin script, Greek script, and so on. Scripts usually span varied parts of several blocks. For further information see [Unicode::UCD](#).

The Unicode code points are just abstract numbers. To input and output these abstract numbers, the numbers must be *encoded* somehow. Unicode defines several *character encoding forms*, of which *UTF-8* is perhaps the most popular. UTF-8 is a variable length encoding that encodes Unicode characters as 1 to 6 bytes (only 4 with the currently defined characters). Other encodings include UTF-16 and UTF-32 and their big- and little-endian variants (UTF-8 is byte-order independent) The ISO/IEC 10646 defines the UCS-2 and UCS-4 encoding forms.

For more information about encodings—for instance, to learn what *surrogates* and *byte order marks* (BOMs) are—see [perlunicode](#).

Perl's Unicode Support

Starting from Perl 5.6.0, Perl has had the capacity to handle Unicode natively. Perl 5.8.0, however, is the first recommended release for serious Unicode work. The maintenance release 5.6.1 fixed many of the problems of the initial Unicode implementation, but for example regular expressions still do not work with Unicode in 5.6.1.

Starting from Perl 5.8.0, the use of `use utf8` is no longer necessary. In earlier releases the `utf8` pragma was used to declare that operations in the current block or file would be Unicode-aware. This model was found to be wrong, or at least clumsy: the "Unicodeness" is now carried with the data, instead of being attached to the operations. Only one case remains where an explicit `use utf8` is needed: if your Perl script itself is encoded in UTF-8, you can use UTF-8 in your identifier names, and in string and regular expression literals, by saying `use utf8`. This is not the default because scripts with legacy 8-bit data in them would break. See [utf8](#).

Perl's Unicode Model

Perl supports both pre-5.6 strings of eight-bit native bytes, and strings of Unicode characters. The principle is that Perl tries to keep its data as eight-bit bytes for as long as possible, but as soon as Unicodeness cannot be avoided, the data is transparently upgraded to Unicode.

Internally, Perl currently uses either whatever the native eight-bit character set of the platform (for example Latin-1) is, defaulting to UTF-8, to encode Unicode strings. Specifically, if all code points in the string are 0xFF or less, Perl uses the native eight-bit character set. Otherwise, it uses UTF-8.

A user of Perl does not normally need to know nor care how Perl happens to encode its internal strings, but it becomes relevant when outputting Unicode strings to a stream without a PerlIO layer — one with the "default" encoding. In such a case, the raw bytes used internally (the native character set or UTF-8, as appropriate for each string) will be used, and a "Wide character" warning will be issued if those strings contain a character beyond 0x00FF.

For example,

```
perl -e 'print "\x{DF}\n", "\x{0100}\x{DF}\n"
```

produces a fairly useless mixture of native bytes and UTF-8, as well as a warning:

```
Wide character in print at ...
```


To output UTF-8, use the `:utf8` output layer. Prepending

```
binmode(STDOUT, ":utf8");
```

to this sample program ensures that the output is completely UTF-8, and removes the program's warning.

If your locale environment variables (`LANGUAGE`, `LC_ALL`, `LC_CTYPE`, `LANG`) contain the strings 'UTF-8' or 'UTF8', regardless of case, then the default encoding of your `STDIN`, `STDOUT`, and `STDERR` and of **any subsequent file open**, is UTF-8. Note that this means that Perl expects other software to work, too: if Perl has been led to believe that `STDIN` should be UTF-8, but then `STDIN` coming in from another command is not UTF-8, Perl will complain about the malformed UTF-8.

All features that combine Unicode and I/O also require using the new PerlIO feature. Almost all Perl 5.8 platforms do use PerlIO, though: you can see whether yours is by running `"perl -V"` and looking for `useperlio=define`.

Unicode and EBCDIC

Perl 5.8.0 also supports Unicode on EBCDIC platforms. There, Unicode support is somewhat more complex to implement since additional conversions are needed at every step. Some problems remain, see [perlebcdic](#) for details.

In any case, the Unicode support on EBCDIC platforms is better than in the 5.6 series, which didn't work much at all for EBCDIC platform. On EBCDIC platforms, the internal Unicode encoding form is UTF-EBCDIC instead of UTF-8. The difference is that as UTF-8 is "ASCII-safe" in that ASCII characters encode to UTF-8 as-is, while UTF-EBCDIC is "EBCDIC-safe".

Creating Unicode

To create Unicode characters in literals for code points above `0xFF`, use the `\x{...}` notation in double-quoted strings:

```
my $smiley = "\x{263a}";
```

Similarly, it can be used in regular expression literals

```
$smiley =~ /\x{263a}/;
```

At run-time you can use `chr()`:

```
my $hebrew_alef = chr(0x05d0);
```

See [perlunicode](#) for how to find all these numeric codes.

Naturally, `ord()` will do the reverse: it turns a character into a code point.

Note that `\x..` (no `{}` and only two hexadecimal digits), `\x{...}`, and `chr(...)` for arguments less than `0x100` (decimal 256) generate an eight-bit character for backward compatibility with older Perls. For arguments of `0x100` or more, Unicode characters are always produced. If you want to force the production of Unicode characters regardless of the numeric value, use `pack("U", ...)` instead of `\x..`, `\x{...}`, or `chr()`.

You can also use the `chardnames` pragma to invoke characters by name in double-quoted strings:

```
use chardnames ':full';
my $arabic_alef = "\N{ARABIC LETTER ALEF}";
```

And, as mentioned above, you can also `pack()` numbers into Unicode characters:

```
my $georgian_an = pack("U", 0x10a0);
```

Note that both `\x{...}` and `\N{...}` are compile-time string constants: you cannot use variables in them. If you want similar run-time functionality, use `chr()` and `chardnames::vianame()`.

Also note that if all the code points for `pack "U"` are below `0x100`, bytes will be generated, just like if you were using `chr()`.

```
my $bytes = pack("U*", 0x80, 0xFF);
```

If you want to force the result to Unicode characters, use the special "U0" prefix. It consumes no arguments but forces the result to be in Unicode characters, instead of bytes.

```
my $chars = pack("U0U*", 0x80, 0xFF);
```

Handling Unicode

Handling Unicode is for the most part transparent: just use the strings as usual. Functions like `index()`, `length()`, and `substr()` will work on the Unicode characters; regular expressions will work on the Unicode characters (see [perlunicode](#) and [perlretut](#)).

Note that Perl considers combining character sequences to be characters, so for example

```
use charnames ':full';
print length("\N{LATIN CAPITAL LETTER A}\N{COMBINING ACUTE ACCENT}"), "\n";
```

will print 2, not 1. The only exception is that regular expressions have `\X` for matching a combining character sequence.

Life is not quite so transparent, however, when working with legacy encodings, I/O, and certain special cases:

Legacy Encodings

When you combine legacy data and Unicode the legacy data needs to be upgraded to Unicode. Normally ISO 8859-1 (or EBCDIC, if applicable) is assumed. You can override this assumption by using the encoding pragma, for example

```
use encoding 'latin2'; # ISO 8859-2
```

in which case literals (string or regular expressions), `chr()`, and `ord()` in your whole script are assumed to produce Unicode characters from ISO 8859-2 code points. Note that the matching for encoding names is forgiving: instead of `latin2` you could have said `Latin 2`, or `iso8859-2`, or other variations. With just

```
use encoding;
```

the environment variable `PERL_ENCODING` will be consulted. If that variable isn't set, the encoding pragma will fail.

The Encode module knows about many encodings and has interfaces for doing conversions between those encodings:

```
use Encode 'from_to';
from_to($data, "iso-8859-3", "utf-8"); # from legacy to utf-8
```

Unicode I/O

Normally, writing out Unicode data

```
print FH $some_string_with_unicode, "\n";
```

produces raw bytes that Perl happens to use to internally encode the Unicode string. Perl's internal encoding depends on the system as well as what characters happen to be in the string at the time. If any of the characters are at code points 0x100 or above, you will get a warning. To ensure that the output is explicitly rendered in the encoding you desire—and to avoid the warning—open the stream with the desired encoding. Some examples:

```
open FH, ">:utf8", "file";

open FH, ">:encoding(ucs2)",      "file";
open FH, ">:encoding(UTF-8)",     "file";
open FH, ">:encoding(shift_jis)", "file";
```

and on already open streams, use `binmode()`:

```
binmode(STDOUT, ":utf8");

binmode(STDOUT, ":encoding(ucs2)");
binmode(STDOUT, ":encoding(UTF-8)");
binmode(STDOUT, ":encoding(shift_jis)");
```

The matching of encoding names is loose: case does not matter, and many encodings have several aliases. Note that the `:utf8` layer must always be specified exactly like that; it is *not* subject to the loose matching of encoding names.

See [PerlIO](#) for the `:utf8` layer, [PerlIO::encoding](#) and [Encode::PerlIO](#) for the `:encoding()` layer, and [Encode::Supported](#) for many encodings supported by the `Encode` module.

Reading in a file that you know happens to be encoded in one of the Unicode or legacy encodings does not magically turn the data into Unicode in Perl's eyes. To do that, specify the appropriate layer when opening files

```
open(my $fh, '<:utf8', 'anything');
my $line_of_unicode = <$fh>;

open(my $fh, '<:encoding(Big5)', 'anything');
my $line_of_unicode = <$fh>;
```

The I/O layers can also be specified more flexibly with the `open` pragma. See [open](#), or look at the following example.

```
use open ':utf8'; # input and output default layer will be UTF-8
open X, ">file";
print X chr(0x100), "\n";
close X;
open Y, "<file";
printf "%#x\n", ord(<Y>); # this should print 0x100
close Y;
```

With the `open` pragma you can use the `:locale` layer

```
$ENV{LC_ALL} = $ENV{LANG} = 'ru_RU.KOI8-R';
# the :locale will probe the locale environment variables like LC_ALL
use open OUT => ':locale'; # russki parusski
open(O, ">koi8");
print O chr(0x430); # Unicode CYRILLIC SMALL LETTER A = KOI8-R 0xc1
close O;
open(I, "<koi8");
printf "%#x\n", ord(<I>), "\n"; # this should print 0xc1
close I;
```

or you can also use the `:encoding(...)` layer

```
open(my $epic, '<:encoding(iso-8859-7)', 'iliad.greek');
my $line_of_unicode = <$epic>;
```

These methods install a transparent filter on the I/O stream that converts data from the specified encoding when it is read in from the stream. The result is always Unicode.

The [open](#) pragma affects all the `open()` calls after the pragma by setting default layers. If you want to affect only certain streams, use explicit layers directly in the `open()` call.

You can switch encodings on an already opened stream by using `binmode()`; see [binmode](#).

The `:locale` does not currently (as of Perl 5.8.0) work with `open()` and `binmode()`, only with the `open` pragma. The `:utf8` and `:encoding(...)` methods do work with all of `open()`, `binmode()`,

and the `open` pragma.

Similarly, you may use these I/O layers on output streams to automatically convert Unicode to the specified encoding when it is written to the stream. For example, the following snippet copies the contents of the file "text.jis" (encoded as ISO-2022-JP, aka JIS) to the file "text.utf8", encoded as UTF-8:

```
open(my $nihongo, '<:encoding(iso2022-jp)', 'text.jis');
open(my $unicode, '>:utf8', 'text.utf8');
while (<$nihongo>) { print $unicode }
```

The naming of encodings, both by the `open()` and by the `open` pragma, is similar to the encoding pragma in that it allows for flexible names: `koi8-r` and `KOI8R` will both be understood.

Common encodings recognized by ISO, MIME, IANA, and various other standardisation organisations are recognised; for a more detailed list see [Encode::Supported](#).

`read()` reads characters and returns the number of characters. `seek()` and `tell()` operate on byte counts, as do `sysread()` and `sysseek()`.

Notice that because of the default behaviour of not doing any conversion upon input if there is no default layer, it is easy to mistakenly write code that keeps on expanding a file by repeatedly encoding the data:

```
# BAD CODE WARNING
open F, "file";
local $/; ## read in the whole file of 8-bit characters
$t = <F>;
close F;
open F, ">:utf8", "file";
print F $t; ## convert to UTF-8 on output
close F;
```

If you run this code twice, the contents of the *file* will be twice UTF-8 encoded. A use `open `:utf8`` would have avoided the bug, or explicitly opening also the *file* for input as UTF-8.

NOTE: the `:utf8` and `:encoding` features work only if your Perl has been built with the new PerlIO feature.

Displaying Unicode As Text

Sometimes you might want to display Perl scalars containing Unicode as simple ASCII (or EBCDIC) text. The following subroutine converts its argument so that Unicode characters with code points greater than 255 are displayed as `\x{...}`, control characters (like `\n`) are displayed as `\x..`, and the rest of the characters as themselves:

```
sub nice_string {
    join("",
        map { $_ > 255 ?                # if wide character...
            sprintf("\x{%04X}", $_) :   # \x{...}
            chr($_) =~ /[[:cntrl:]]/?    # else if control character ...
            sprintf("\x{%02X}", $_) :    # \x..
            chr($_)                      # else as themselves
        } unpack("U*", $_[0])));
    # unpack Unicode characters
}
```

For example,

```
nice_string("foo\x{100}bar\n")
```

returns:

```
"foo\x{0100}bar\x0A"
```

Special Cases

- Bit Complement Operator `~` And `vec()`

The bit complement operator `~` may produce surprising results if used on strings containing characters with ordinal values above 255. In such a case, the results are consistent with the internal encoding of the characters, but not with much else. So don't do that. Similarly for `vec()`: you will be operating on the internally-encoded bit patterns of the Unicode characters, not on the code point values, which is very probably not what you want.

- Peeking At Perl's Internal Encoding

Normal users of Perl should never care how Perl encodes any particular Unicode string (because the normal ways to get at the contents of a string with Unicode—via input and output—should always be via explicitly-defined I/O layers). But if you must, there are two ways of looking behind the scenes.

One way of peeking inside the internal encoding of Unicode characters is to use `unpack("C*", ...)` to get the bytes or `unpack("H*", ...)` to display the bytes:

```
# this prints  c4 80  for the UTF-8 bytes 0xc4 0x80
print join(" ", unpack("H*", pack("U", 0x100))), "\n";
```

Yet another way would be to use the `Devel::Peek` module:

```
perl -MDevel::Peek -e 'Dump(chr(0x100))'
```

That shows the UTF8 flag in `FLAGS` and both the UTF-8 bytes and Unicode characters in `PV`. See also later in this document the discussion about the `is_utf8` function of the `Encode` module.

Advanced Topics

- String Equivalence

The question of string equivalence turns somewhat complicated in Unicode: what do you mean by "equal"?

(Is LATIN CAPITAL LETTER A WITH ACUTE equal to LATIN CAPITAL LETTER A?)

The short answer is that by default Perl compares equivalence (`eq`, `ne`) based only on code points of the characters. In the above case, the answer is no (because `0x00C1` != `0x0041`). But sometimes, any CAPITAL LETTER As should be considered equal, or even As of any case.

The long answer is that you need to consider character normalization and casing issues: see [Unicode::Normalize](#), Unicode Technical Reports #15 and #21, *Unicode Normalization Forms and Case Mappings*, <http://www.unicode.org/unicode/reports/tr15/> and <http://www.unicode.org/unicode/reports/tr21/>

As of Perl 5.8.0, the "Full" case-folding of *Case Mappings/SpecialCasing* is implemented.

- String Collation

People like to see their strings nicely sorted—or as Unicode parlance goes, collated. But again, what do you mean by collate?

(Does LATIN CAPITAL LETTER A WITH ACUTE come before or after LATIN CAPITAL LETTER A WITH GRAVE?)

The short answer is that by default, Perl compares strings (`lt`, `le`, `cmp`, `ge`, `gt`) based only on the code points of the characters. In the above case, the answer is "after", since `0x00C1` `0x00C0`.

The long answer is that "it depends", and a good answer cannot be given without knowing (at the very least) the language context. See [Unicode::Collate](#), and *Unicode Collation Algorithm* <http://www.unicode.org/unicode/reports/tr10/>

Miscellaneous

- Character Ranges and Classes

Character ranges in regular expression character classes (`/[a-z]/`) and in the `tr///` (also known as `y///`) operator are not magically Unicode-aware. What this means that `[A-Za-z]` will not magically start to mean "all alphabetic letters"; not that it does mean that even for 8-bit characters, you should be using `/[:alpha:]/` in that case.

For specifying character classes like that in regular expressions, you can use the various Unicode properties—`\pL`, or perhaps `\p{Alphabetic}`, in this particular case. You can use Unicode code points as the end points of character ranges, but there is no magic associated with specifying a certain range. For further information—there are dozens of Unicode character classes—see [perlunicode](#).

- String-To-Number Conversions

Unicode does define several other decimal—and numeric—characters besides the familiar 0 to 9, such as the Arabic and Indic digits. Perl does not support string-to-number conversion for digits other than ASCII 0 to 9 (and ASCII a to f for hexadecimal).

Questions With Answers

- Will My Old Scripts Break?

Very probably not. Unless you are generating Unicode characters somehow, old behaviour should be preserved. About the only behaviour that has changed and which could start generating Unicode is the old behaviour of `chr()` where supplying an argument more than 255 produced a character modulo 255. `chr(300)`, for example, was equal to `chr(45)` or `"-"` (in ASCII), now it is LATIN CAPITAL LETTER I WITH BREVE.

- How Do I Make My Scripts Work With Unicode?

Very little work should be needed since nothing changes until you generate Unicode data. The most important thing is getting input as Unicode; for that, see the earlier I/O discussion.

- How Do I Know Whether My String Is In Unicode?

You shouldn't care. No, you really shouldn't. No, really. If you have to care—beyond the cases described above—it means that we didn't get the transparency of Unicode quite right.

Okay, if you insist:

```
use Encode 'is_utf8';
print is_utf8($string) ? 1 : 0, "\n";
```

But note that this doesn't mean that any of the characters in the string are necessary UTF-8 encoded, or that any of the characters have code points greater than 0xFF (255) or even 0x80 (128), or that the string has any characters at all. All the `is_utf8()` does is to return the value of the internal "utf8ness" flag attached to the `$string`. If the flag is off, the bytes in the scalar are interpreted as a single byte encoding. If the flag is on, the bytes in the scalar are interpreted as the (multi-byte, variable-length) UTF-8 encoded code points of the characters. Bytes added to an UTF-8 encoded string are automatically upgraded to UTF-8. If mixed non-UTF8 and UTF-8 scalars are merged (double-quoted interpolation, explicit concatenation, and `printf/sprintf` parameter substitution), the result will be UTF-8 encoded as if copies of the byte strings were upgraded to UTF-8: for example,

```
$a = "ab\x80c";
$b = "\x{100}";
print "$a = $b\n";
```

the output string will be UTF-8-encoded `ab\x80c\x{100}\n`, but note that `$a` will stay byte-encoded.

Sometimes you might really need to know the byte length of a string instead of the character length. For that use either the `Encode::encode_utf8()` function or the `bytes` pragma and its only defined function `length()`:

```
my $unicode = chr(0x100);
print length($unicode), "\n"; # will print 1
require Encode;
print length(Encode::encode_utf8($unicode)), "\n"; # will print 2
use bytes;
print length($unicode), "\n"; # will also print 2
# (the 0xC4 0x80 of the UTF-8)
```

- How Do I Detect Data That's Not Valid In a Particular Encoding?

Use the `Encode` package to try converting it. For example,

```
use Encode 'encode_utf8';
if (encode_utf8($string_of_bytes_that_I_think_is_utf8)) {
    # valid
} else {
    # invalid
}
```

For UTF-8 only, you can use:

```
use warnings;
@chars = unpack("U0U*", $string_of_bytes_that_I_think_is_utf8);
```

If invalid, a Malformed UTF-8 character (byte 0x##) in `unpack` warning is produced. The "U0" means "expect strictly UTF-8 encoded Unicode". Without that the `unpack("U*", ...)` would accept also data like `chr(0xFF)`, similarly to the `pack` as we saw earlier.

- How Do I Convert Binary Data Into a Particular Encoding, Or Vice Versa?

This probably isn't as useful as you might think. Normally, you shouldn't need to.

In one sense, what you are asking doesn't make much sense: encodings are for characters, and binary data are not "characters", so converting "data" into some encoding isn't meaningful unless you know in what character set and encoding the binary data is in, in which case it's not just binary data, now is it?

If you have a raw sequence of bytes that you know should be interpreted via a particular encoding, you can use `Encode`:

```
use Encode 'from_to';
from_to($data, "iso-8859-1", "utf-8"); # from latin-1 to utf-8
```

The call to `from_to()` changes the bytes in `$data`, but nothing material about the nature of the string has changed as far as Perl is concerned. Both before and after the call, the string `$data` contains just a bunch of 8-bit bytes. As far as Perl is concerned, the encoding of the string remains as "system-native 8-bit bytes".

You might relate this to a fictional 'Translate' module:

```
use Translate;
my $phrase = "Yes";
Translate::from_to($phrase, 'english', 'deutsch');
## phrase now contains "Ja"
```

The contents of the string changes, but not the nature of the string. Perl doesn't know any more after the call than before that the contents of the string indicates the affirmative.

Back to converting data. If you have (or want) data in your system's native 8-bit encoding (e.g. Latin-1, EBCDIC, etc.), you can use `pack/unpack` to convert to/from Unicode.

```
$native_string = pack("C*", unpack("U*", $Unicode_string));
$Unicode_string = pack("U*", unpack("C*", $native_string));
```

If you have a sequence of bytes you **know** is valid UTF-8, but Perl doesn't know it yet, you can make Perl a believer, too:

```
use Encode 'decode_utf8';
$Unicode = decode_utf8($bytes);
```

You can convert well-formed UTF-8 to a sequence of bytes, but if you just want to convert random binary data into UTF-8, you can't. **Any random collection of bytes isn't well-formed UTF-8.** You can use `unpack("C*", $string)` for the former, and you can create well-formed Unicode data by `pack("U*", 0xff, ...)`.

- How Do I Display Unicode? How Do I Input Unicode?

See <http://www.alanwood.net/unicode/> and <http://www.cl.cam.ac.uk/~mgk25/unicode.html>

- How Does Unicode Work With Traditional Locales?

In Perl, not very well. Avoid using locales through the `locale` pragma. Use only one or the other.

Hexadecimal Notation

The Unicode standard prefers using hexadecimal notation because that more clearly shows the division of Unicode into blocks of 256 characters. Hexadecimal is also simply shorter than decimal. You can use decimal notation, too, but learning to use hexadecimal just makes life easier with the Unicode standard. The `U+HHHH` notation uses hexadecimal, for example.

The `0x` prefix means a hexadecimal number, the digits are 0–9 *and* a–f (or A–F, case doesn't matter). Each hexadecimal digit represents four bits, or half a byte. `print 0x...`, `"\n"` will show a hexadecimal number in decimal, and `printf "%x\n", $decimal` will show a decimal number in hexadecimal. If you have just the "hex digits" of a hexadecimal number, you can use the `hex()` function.

```
print 0x0009, "\n";    # 9
print 0x000a, "\n";    # 10
print 0x000f, "\n";    # 15
print 0x0010, "\n";    # 16
print 0x0011, "\n";    # 17
print 0x0100, "\n";    # 256

print 0x0041, "\n";    # 65

printf "%x\n", 65;     # 41
printf "%#x\n", 65;    # 0x41

print hex("41"), "\n"; # 65
```

Further Resources

- Unicode Consortium

<http://www.unicode.org/>

- Unicode FAQ

<http://www.unicode.org/unicode/faq/>

- Unicode Glossary

<http://www.unicode.org/glossary/>

- Unicode Useful Resources
`http://www.unicode.org/unicode/onlinedat/resources.html`
- Unicode and Multilingual Support in HTML, Fonts, Web Browsers and Other Applications
`http://www.alanwood.net/unicode/`
- UTF-8 and Unicode FAQ for Unix/Linux
`http://www.cl.cam.ac.uk/~mgk25/unicode.html`
- Legacy Character Sets
`http://www.czyborra.com/`
`http://www.eki.ee/letter/`
- The Unicode support files live within the Perl installation in the directory
`$Config{installprivlib}/unicore`
in Perl 5.8.0 or newer, and
`$Config{installprivlib}/unicode`
in the Perl 5.6 series. (The renaming to *lib/unicore* was done to avoid naming conflicts with *lib/Unicode* in case-insensitive filesystems.) The main Unicode data file is *UnicodeData.txt* (or *Unicode.301* in Perl 5.6.1.) You can find the `$Config{installprivlib}` by

```
perl "-V:installprivlib"
```


You can explore various information from the Unicode data files using the `Unicode::UCD` module.

UNICODE IN OLDER PERLS

If you cannot upgrade your Perl to 5.8.0 or later, you can still do some Unicode processing by using the modules `Unicode::String`, `Unicode::Map8`, and `Unicode::Map`, available from CPAN. If you have the GNU recode installed, you can also use the Perl front-end `Convert::Recode` for character conversions.

The following are fast conversions from ISO 8859-1 (Latin-1) bytes to UTF-8 bytes, the code works even with older Perl 5 versions.

```
# ISO 8859-1 to UTF-8
s/( [\x80-\xFF] ) /chr(0xC0|ord($1)>>6).chr(0x80|ord($1)&0x3F)/eg;

# UTF-8 to ISO 8859-1
s/( [\xC2-\xC3] ) ( [\x80-\xBF] ) /chr(ord($1)<<6&0xC0|ord($2)&0x3F)/eg;
```

SEE ALSO

[perlunicode](#), [Encode](#), [encoding](#), [open](#), [utf8](#), [bytes](#), [perlretut](#), [Unicode::Collate](#), [Unicode::Normalize](#), [Unicode::UCD](#)

ACKNOWLEDGMENTS

Thanks to the kind readers of the `perl5-porters@perl.org`, `perl-unicode@perl.org`, `linux-utf8@nl.linux.org`, and `unicore@unicode.org` mailing lists for their valuable feedback.

AUTHOR, COPYRIGHT, AND LICENSE

Copyright 2001–2002 Jarkko Hietaniemi <jhi@iki.fi>

This document may be distributed under the same terms as Perl itself.

NAME

perlutil – utilities packaged with the Perl distribution

DESCRIPTION

Along with the Perl interpreter itself, the Perl distribution installs a range of utilities on your system. There are also several utilities which are used by the Perl distribution itself as part of the install process. This document exists to list all of these utilities, explain what they are for and provide pointers to each module's documentation, if appropriate.

DOCUMENTATION*perldoc|perldoc*

The main interface to Perl's documentation is *perldoc*, although if you're reading this, it's more than likely that you've already found it. *perldoc* will extract and format the documentation from any file in the current directory, any Perl module installed on the system, or any of the standard documentation pages, such as this one. Use *perldoc* <name> to get information on any of the utilities described in this document.

pod2man|pod2man and *pod2text|pod2text*

If it's run from a terminal, *perldoc* will usually call *pod2man* to translate POD (Plain Old Documentation – see *perlpod* for an explanation) into a manpage, and then run *man* to display it; if *man* isn't available, *pod2text* will be used instead and the output piped through your favourite pager.

pod2html|pod2html and *pod2latex|pod2latex*

As well as these two, there are two other converters: *pod2html* will produce HTML pages from POD, and *pod2latex*, which produces LaTeX files.

pod2usage|pod2usage

If you just want to know how to use the utilities described here, *pod2usage* will just extract the "USAGE" section; some of the utilities will automatically call *pod2usage* on themselves when you call them with *-help*.

podselect|podselect

pod2usage is a special case of *podselect*, a utility to extract named sections from documents written in POD. For instance, while utilities have "USAGE" sections, Perl modules usually have "SYNOPSIS" sections: *podselect -s "SYNOPSIS" ...* will extract this section for a given file.

podchecker|podchecker

If you're writing your own documentation in POD, the *podchecker* utility will look for errors in your markup.

splain|splain

splain is an interface to *perldiag* – paste in your error message to it, and it'll explain it for you.

roffitall|roffitall

The *roffitall* utility is not installed on your system but lives in the *pod/* directory of your Perl source kit; it converts all the documentation from the distribution to **roff* format, and produces a typeset PostScript or text file of the whole lot.

CONVERTORS

To help you convert legacy programs to Perl, we've included three conversion filters:

a2p|a2p

a2p converts *awk* scripts to Perl programs; for example, *a2p -F:* on the simple *awk* script `{print $2}` will produce a Perl program based around this code:

```
while (<>) {
    ($F1d1,$F1d2) = split(/[:\n]/, $_, 9999);
```

```
        print $Fld2;
    }
}
```

[s2p/s2p](#)

Similarly, *s2p* converts *sed* scripts to Perl programs. *s2p* run on `s/foo/bar` will produce a Perl program based around this:

```
while (<>) {
    chomp;
    s/foo/bar/g;
    print if $printit;
}
```

[find2perl/find2perl](#)

Finally, *find2perl* translates *find* commands to Perl equivalents which use the [File::Find/File::Find](#) module. As an example, `find2perl . -user root -perm 4000 -print` produces the following callback subroutine for `File::Find`:

```
sub wanted {
    my ($dev,$ino,$mode,$nlink,$uid,$gid);
    (($dev,$ino,$mode,$nlink,$uid,$gid) = lstat($_)) &&
    $uid == $uid{'root'}) &&
    (($mode & 0777) == 04000);
    print("$name\n");
}
```

As well as these filters for converting other languages, the [pl2pm/pl2pm](#) utility will help you convert old-style Perl 4 libraries to new-style Perl5 modules.

Administration

[libnetcfg/libnetcfg](#)

To display and change the libnet configuration run the `libnetcfg` command.

Development

There are a set of utilities which help you in developing Perl programs, and in particular, extending Perl with C.

[perlbug/perlbug](#)

perlbug is the recommended way to report bugs in the perl interpreter itself or any of the standard library modules back to the developers; please read through the documentation for *perlbug* thoroughly before using it to submit a bug report.

[h2ph/h2ph](#)

Back before Perl had the XS system for connecting with C libraries, programmers used to get library constants by reading through the C header files. You may still see `require 'syscall.ph'` or similar around – the *.ph* file should be created by running *h2ph* on the corresponding *.h* file. See the *h2ph* documentation for more on how to convert a whole bunch of header files at once.

[c2ph/c2ph](#) and [pstruct/pstruct](#)

c2ph and *pstruct*, which are actually the same program but behave differently depending on how they are called, provide another way of getting at C with Perl – they'll convert C structures and union declarations to Perl code. This is deprecated in favour of *h2xs* these days.

[h2xs/h2xs](#)

h2xs converts C header files into XS modules, and will try and write as much glue between C libraries and Perl modules as it can. It's also very useful for creating skeletons of pure Perl modules.

[*dprofpp/dprofpp*](#)

Perl comes with a profiler, the *Devel::DProf* module. The *dprofpp* utility analyzes the output of this profiler and tells you which subroutines are taking up the most run time. See [*Devel::DProf*](#) for more information.

[*perlcc/perlcc*](#)

perlcc is the interface to the experimental Perl compiler suite.

SEE ALSO

[*perldoc/perldoc*](#), [*pod2man/pod2man*](#), [*perlpod*](#), [*pod2html/pod2html*](#), [*pod2usage/pod2usage*](#),
[*podselect/podselect*](#), [*podchecker/podchecker*](#), [*splain/splain*](#), [*perldiag*](#), [*roffitall/roffitall*](#), [*a2p/a2p*](#), [*s2p/s2p*](#),
[*find2perl/find2perl*](#), [*File::Find/File::Find*](#), [*pl2pm/pl2pm*](#), [*perlbug/perlbug*](#), [*h2ph/h2ph*](#), [*c2ph/c2ph*](#), [*h2xs/h2xs*](#),
[*dprofpp/dprofpp*](#), [*Devel::DProf*](#), [*perlcc/perlcc*](#)

NAME

perlvar – Perl predefined variables

DESCRIPTION

Predefined Names

The following names have special meaning to Perl. Most punctuation names have reasonable mnemonics, or analogs in the shells. Nevertheless, if you wish to use long variable names, you need only say

```
use English;
```

at the top of your program. This aliases all the short names to the long names in the current package. Some even have medium names, generally borrowed from **awk**. In general, it's best to use the

```
use English '-no_match_vars';
```

invocation if you don't need `$PREMATCH`, `$MATCH`, or `$POSTMATCH`, as it avoids a certain performance hit with the use of regular expressions. See [English](#).

Variables that depend on the currently selected filehandle may be set by calling an appropriate object method on the `IO::Handle` object, although this is less efficient than using the regular built-in variables. (Summary lines below for this contain the word `HANDLE`.) First you must say

```
use IO::Handle;
```

after which you may use either

```
method HANDLE EXPR
```

or more safely,

```
HANDLE->method(EXPR)
```

Each method returns the old value of the `IO::Handle` attribute. The methods each take an optional `EXPR`, which, if supplied, specifies the new value for the `IO::Handle` attribute in question. If not supplied, most methods do nothing to the current value—except for `autoflush()`, which will assume a 1 for you, just to be different.

Because loading in the `IO::Handle` class is an expensive operation, you should learn how to use the regular built-in variables.

A few of these variables are considered "read-only". This means that if you try to assign to this variable, either directly or indirectly through a reference, you'll raise a run-time exception.

You should be very careful when modifying the default values of most special variables described in this document. In most cases you want to localize these variables before changing them, since if you don't, the change may affect other modules which rely on the default values of the special variables that you have changed. This is one of the correct ways to read the whole file at once:

```
open my $fh, "foo" or die $!;
local $/; # enable localized slurp mode
my $content = <$fh>;
close $fh;
```

But the following code is quite bad:

```
open my $fh, "foo" or die $!;
undef $/; # enable slurp mode
my $content = <$fh>;
close $fh;
```

since some other module, may want to read data from some file in the default "line mode", so if the code we have just presented has been executed, the global value of `$/` is now changed for any other code running inside the same Perl interpreter.

Usually when a variable is localized you want to make sure that this change affects the shortest scope possible. So unless you are already inside some short `{ }` block, you should create one yourself. For example:

```
my $content = '';
open my $fh, "foo" or die $!;
{
    local $/;
    $content = <$fh>;
}
close $fh;
```

Here is an example of how your own code can go broken:

```
for (1..5){
    nasty_break();
    print "$_ ";
}
sub nasty_break {
    $_ = 5;
    # do something with $_
}
```

You probably expect this code to print:

```
1 2 3 4 5
```

but instead you get:

```
5 5 5 5 5
```

Why? Because `nasty_break()` modifies `$_` without localizing it first. The fix is to add `local()`:

```
local $_ = 5;
```

It's easy to notice the problem in such a short example, but in more complicated code you are looking for trouble if you don't localize changes to the special variables.

The following list is ordered by scalar variables first, then the arrays, then the hashes.

`$ARG`

`$_` The default input and pattern-searching space. The following pairs are equivalent:

```
while (<>) {...}      # equivalent only in while!
while (defined($_ = <>)) {...}

/^Subject:/
$_ =~ /^Subject:/

tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/

chomp
chomp($_)
```

Here are the places where Perl will assume `$_` even if you don't use it:

- Various unary functions, including functions like `ord()` and `int()`, as well as the all file tests (`-f`, `-d`) except for `-t`, which defaults to STDIN.
- Various list functions like `print()` and `unlink()`.

- The pattern matching operations `m//`, `s///`, and `tr///` when used without an `=~` operator.
- The default iterator variable in a `foreach` loop if no other variable is supplied.
- The implicit iterator variable in the `grep()` and `map()` functions.
- The default place to put an input record when a `<<FH` operation's result is tested by itself as the sole criterion of a `while` test. Outside a `while` test, this will not happen.

(Mnemonic: underline is understood in certain operations.)

`$a`

`$b` Special package variables when using `sort()`, see [sort](#). Because of this specialness `$a` and `$b` don't need to be declared (using `local()`, `use vars`, or `our()`) even when using the strict vars pragma. Don't lexicalize them with `my $a` or `my $b` if you want to be able to use them in the `sort()` comparison block or function.

`$<digits`

Contains the subpattern from the corresponding set of capturing parentheses from the last pattern match, not counting patterns matched in nested blocks that have been exited already. (Mnemonic: like `\digits`.) These variables are all read-only and dynamically scoped to the current BLOCK.

`$MATCH`

`$&` The string matched by the last successful pattern match (not counting any matches hidden within a BLOCK or `eval()` enclosed by the current BLOCK). (Mnemonic: like `&` in some editors.) This variable is read-only and dynamically scoped to the current BLOCK.

The use of this variable anywhere in a program imposes a considerable performance penalty on all regular expression matches. See [BUGS](#).

`$PREMATCH`

`$'` The string preceding whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or `eval` enclosed by the current BLOCK). (Mnemonic: `'` often precedes a quoted string.) This variable is read-only.

The use of this variable anywhere in a program imposes a considerable performance penalty on all regular expression matches. See [BUGS](#).

`$POSTMATCH`

`$'` The string following whatever was matched by the last successful pattern match (not counting any matches hidden within a BLOCK or `eval()` enclosed by the current BLOCK). (Mnemonic: `'` often follows a quoted string.) Example:

```
local $_ = 'abcdefghi';
/def/;
print "$':$&:$'\n";           # prints abc:def:ghi
```

This variable is read-only and dynamically scoped to the current BLOCK.

The use of this variable anywhere in a program imposes a considerable performance penalty on all regular expression matches. See [BUGS](#).

`$LAST_PAREN_MATCH`

`$+` The text matched by the last bracket of the last successful search pattern. This is useful if you don't know which one of a set of alternative patterns matched. For example:

```
/Version: (.*)|Revision: (.*)/ && ($rev = $+);
```

(Mnemonic: be positive and forward looking.) This variable is read-only and dynamically scoped to the current BLOCK.

`$^N` The text matched by the used group most-recently closed (i.e. the group with the rightmost closing parenthesis) of the last successful search pattern. (Mnemonic: the (possibly) Nested parenthesis that most recently closed.)

This is primarily used inside `(?{ ... })` blocks for examining text recently matched. For example, to effectively capture text to a variable (in addition to `$1`, `$2`, etc.), replace `(...)` with

```
(?: (... ) ( ? { $var = $^N } ) )
```

By setting and then using `$var` in this way relieves you from having to worry about exactly which numbered set of parentheses they are.

This variable is dynamically scoped to the current BLOCK.

@LAST_MATCH_END

@+ This array holds the offsets of the ends of the last successful submatches in the currently active dynamic scope. `$+[0]` is the offset into the string of the end of the entire match. This is the same value as what the `pos` function returns when called on the variable that was matched against. The *n*th element of this array holds the offset of the *n*th submatch, so `$+[1]` is the offset past where `$1` ends, `$+[2]` the offset past where `$2` ends, and so on. You can use `$#+` to determine how many subgroups were in the last successful match. See the examples given for the `@-` variable.

\$MULTILINE_MATCHING

`$*` Set to a non-zero integer value to do multi-line matching within a string, 0 (or undefined) to tell Perl that it can assume that strings contain a single line, for the purpose of optimizing pattern matches. Pattern matches on strings containing multiple newlines can produce confusing results when `$*` is 0 or undefined. Default is undefined. (Mnemonic: `*` matches multiple things.) This variable influences the interpretation of only `^` and `$`. A literal newline can be searched for even when `$* == 0`.

Use of `$*` is deprecated in modern Perl, supplanted by the `/s` and `/m` modifiers on pattern matching.

Assigning a non-numerical value to `$*` triggers a warning (and makes `$*` act if `$* == 0`), while assigning a numerical value to `$*` makes that an implicit `int` is applied on the value.

HANDLE-input_line_number(EXPR)

`$INPUT_LINE_NUMBER`

`$NR`

`$.` Current line number for the last filehandle accessed.

Each filehandle in Perl counts the number of lines that have been read from it. (Depending on the value of `$/,` Perl's idea of what constitutes a line may not match yours.) When a line is read from a filehandle (via `readline()` or `< <`), or when `tell()` or `seek()` is called on it, `$.` becomes an alias to the line counter for that filehandle.

You can adjust the counter by assigning to `$.`, but this will not actually move the seek pointer. *Localizing `$.` will not localize the filehandle's line count.* Instead, it will localize perl's notion of which filehandle `$.` is currently aliased to.

`$.` is reset when the filehandle is closed, but **not** when an open filehandle is reopened without an intervening `close()`. For more details, see *"IO Operators"*. Because `< <` never does an explicit close, line numbers increase across ARGV files (but see examples in *eof*).

You can also use `< HANDLE-input_line_number(EXPR)` to access the line counter for a given filehandle without having to worry about which handle you last accessed.

(Mnemonic: many programs use `"."` to mean the current line number.)

IO::Handle

\$INPUT_RECORD_SEPARATOR

\$RS

`$/` The input record separator, newline by default. This influences Perl's idea of what a "line" is. Works like **awk**'s `RS` variable, including treating empty lines as a terminator if set to the null string. (An empty line cannot contain any spaces or tabs.) You may set it to a multi-character string to match a multi-character terminator, or to `undef` to read through the end of file. Setting it to `"\n\n"` means something slightly different than setting to `" "`, if the file contains consecutive empty lines. Setting to `" "` will treat two or more consecutive empty lines as a single empty line. Setting to `"\n\n"` will blindly assume that the next input character belongs to the next paragraph, even if it's a newline. (Mnemonic: `/` delimits line boundaries when quoting poetry.)

```
local $/;           # enable "slurp" mode
local $_ = <FH>;     # whole file now here
s/\n[ \t]+/ /g;
```

Remember: the value of `$/` is a string, not a regex. **awk** has to be better for something. :-)

Setting `$/` to a reference to an integer, scalar containing an integer, or scalar that's convertible to an integer will attempt to read records instead of lines, with the maximum record size being the referenced integer. So this:

```
local $/ = \32768; # or "\"32768", or \"$var_containing_32768
open my $fh, $myfile or die $!;
local $_ = <$fh>;
```

will read a record of no more than 32768 bytes from `FILE`. If you're not reading from a record-oriented file (or your OS doesn't have record-oriented files), then you'll likely get a full chunk of data with every read. If a record is larger than the record size you've set, you'll get the record back in pieces.

On VMS, record reads are done with the equivalent of `sysread`, so it's best not to mix record and non-record reads on the same file. (This is unlikely to be a problem, because any file you'd want to read in record mode is probably unusable in line mode.) Non-VMS systems do normal I/O, so it's safe to mix record and non-record reads of a file.

See also *Newlines in perlport*. Also see `$.`

HANDLE-autoflush(EXPR)

\$OUTPUT_AUTOFLUSH

`$|` If set to nonzero, forces a flush right away and after every write or print on the currently selected output channel. Default is 0 (regardless of whether the channel is really buffered by the system or not; `$|` tells you only whether you've asked Perl explicitly to flush after each write). `STDOUT` will typically be line buffered if output is to the terminal and block buffered otherwise. Setting this variable is useful primarily when you are outputting to a pipe or socket, such as when you are running a Perl program under **rsh** and want to see the output as it's happening. This has no effect on input buffering. See *getc* for that. (Mnemonic: when you want your pipes to be piping hot.)

IO::Handle-output_field_separator EXPR

\$OUTPUT_FIELD_SEPARATOR

\$OFS

`$,` The output field separator for the print operator. Ordinarily the print operator simply prints out its arguments without further adornment. To get behavior more like **awk**, set this variable as you would set **awk**'s `OFS` variable to specify what is printed between fields. (Mnemonic: what is printed when there is a `","` in your print statement.)

IO::Handle-output_record_separator EXPR

\$OUTPUT_RECORD_SEPARATOR

\$ORS

\$\ The output record separator for the print operator. Ordinarily the print operator simply prints out its arguments as is, with no trailing newline or other end-of-record string added. To get behavior more like **awk**, set this variable as you would set **awk**'s ORS variable to specify what is printed at the end of the print. (Mnemonic: you set \$\ instead of adding "\n" at the end of the print. Also, it's just like \$/, but it's what you get "back" from Perl.)

\$LIST_SEPARATOR

\$" This is like \$, except that it applies to array and slice values interpolated into a double-quoted string (or similar interpreted string). Default is a space. (Mnemonic: obvious, I think.)

\$SUBSCRIPT_SEPARATOR

\$SUBSEP

\$; The subscript separator for multidimensional array emulation. If you refer to a hash element as

```
$foo{$a,$b,$c}
```

it really means

```
$foo{join($;, $a, $b, $c)}
```

But don't put

```
@foo{$a,$b,$c}      # a slice--note the @
```

which means

```
($foo{$a},$foo{$b},$foo{$c})
```

Default is "\034", the same as SUBSEP in **awk**. If your keys contain binary data there might not be any safe value for \$;. (Mnemonic: comma (the syntactic subscript separator) is a semi-semicolon. Yeah, I know, it's pretty lame, but \$, is already taken for something more important.)

Consider using "real" multidimensional arrays as described in [perllol](#).

\$OFMT

\$# The output format for printed numbers. This variable is a half-hearted attempt to emulate **awk**'s OFMT variable. There are times, however, when **awk** and Perl have differing notions of what counts as numeric. The initial value is "%ng", where *n* is the value of the macro DBL_DIG from your system's *float.h*. This is different from **awk**'s default OFMT setting of "%.6g", so you need to set \$# explicitly to get **awk**'s value. (Mnemonic: # is the number sign.)

Use of \$# is deprecated.

HANDLE-format_page_number(EXPR)

\$FORMAT_PAGE_NUMBER

% The current page number of the currently selected output channel. Used with formats. (Mnemonic: % is page number in **nroff**.)

HANDLE-format_lines_per_page(EXPR)

\$FORMAT_LINES_PER_PAGE

= The current page length (printable lines) of the currently selected output channel. Default is 60. Used with formats. (Mnemonic: = has horizontal lines.)

HANDLE-format_lines_left(EXPR)

\$FORMAT_LINES_LEFT

- The number of lines left on the page of the currently selected output channel. Used with formats. (Mnemonic: lines_on_page - lines_printed.)

@LAST_MATCH_START

@- `$-[0]` is the offset of the start of the last successful match. `$-[n]` is the offset of the start of the substring matched by *n*-th subpattern, or undef if the subpattern did not match.

Thus after a match against `$_, $&` coincides with `substr $_, $-[0], $+[0] - $-[0]`. Similarly, `$n` coincides with `substr $_, $-[n], $+[n] - $-[n]` if `$-[n]` is defined, and `$+` coincides with `substr $_, $-[$#-], $+[$#-]`. One can use `$#-` to find the last matched subgroup in the last successful match. Contrast with `$#+`, the number of subgroups in the regular expression. Compare with `@+`.

This array holds the offsets of the beginnings of the last successful submatches in the currently active dynamic scope. `$-[0]` is the offset into the string of the beginning of the entire match. The *n*th element of this array holds the offset of the *n*th submatch, so `$-[1]` is the offset where `$1` begins, `$-[2]` the offset where `$2` begins, and so on.

After a match against some variable `$var`:

```
$' is the same as substr($var, 0, $-[0])
$& is the same as substr($var, $-[0], $+[0] - $-[0])
$' is the same as substr($var, $+[0])
$1 is the same as substr($var, $-[1], $+[1] - $-[1])
$2 is the same as substr($var, $-[2], $+[2] - $-[2])
$3 is the same as substr $var, $-[3], $+[3] - $-[3])
```

HANDLE=format_name(EXPR)

`$FORMAT_NAME`

\$~ The name of the current report format for the currently selected output channel. Default is the name of the filehandle. (Mnemonic: brother to `$^.`)

HANDLE=format_top_name(EXPR)

`$FORMAT_TOP_NAME`

\$^ The name of the current top-of-page format for the currently selected output channel. Default is the name of the filehandle with `_TOP` appended. (Mnemonic: points to top of page.)

IO::Handle=format_line_break_characters EXPR

`$FORMAT_LINE_BREAK_CHARACTERS`

\$: The current set of characters after which a string may be broken to fill continuation fields (starting with `^`) in a format. Default is `"\n-`", to break on whitespace or hyphens. (Mnemonic: a "colon" in poetry is a part of a line.)

IO::Handle=format_formfeed EXPR

`$FORMAT_FORMFEED`

\$^L What formats output as a form feed. Default is `\f`.

\$ACCUMULATOR

\$^A The current value of the `write()` accumulator for `format()` lines. A format contains `formline()` calls that put their result into `$^A`. After calling its format, `write()` prints out the contents of `$^A` and empties. So you never really see the contents of `$^A` unless you call `formline()` yourself and then look at it. See [perlform](#) and [Formline\(\)](#).

\$CHILD_ERROR

\$? The status returned by the last pipe close, backtick (```) command, successful call to `wait()` or `waitpid()`, or from the `system()` operator. This is just the 16-bit status word returned by the `wait()` system call (or else is made up to look like it). Thus, the exit value of the subprocess is really `(<< $? 8)`, and `$? & 127` gives which signal, if any, the process died from, and `$? & 128` reports whether there was a core dump. (Mnemonic: similar to **sh** and **ksh**.)

Additionally, if the `h_errno` variable is supported in C, its value is returned via `$?` if any `gethost*()` function fails.

If you have installed a signal handler for SIGCHLD, the value of `$?` will usually be wrong outside that handler.

Inside an `END` subroutine `$?` contains the value that is going to be given to `exit()`. You can modify `$?` in an `END` subroutine to change the exit status of your program. For example:

```
END {
    $? = 1 if $? == 255; # die would make it 255
}
```

Under VMS, the pragma `use vmsish 'status'` makes `$?` reflect the actual VMS exit status, instead of the default emulation of POSIX status; see [\\$?](#) for details.

Also see [Error Indicators](#).

`${^ENCODING}`

The *object reference* to the Encode object that is used to convert the source code to Unicode. Thanks to this variable your perl script does not have to be written in UTF-8. Default is *undef*. The direct manipulation of this variable is highly discouraged. See [encoding](#) for more details.

`$OS_ERROR`

`$ERRNO`

`$!` If used numerically, yields the current value of the C `errno` variable, or in other words, if a system or library call fails, it sets this variable. This means that the value of `$!` is meaningful only *immediately* after a **failure**:

```
if (open(FH, $filename)) {
    # Here $! is meaningless.
    ...
} else {
    # ONLY here is $! meaningful.
    ...
    # Already here $! might be meaningless.
}
# Since here we might have either success or failure,
# here $! is meaningless.
```

In the above *meaningless* stands for anything: zero, non-zero, *undef*. A successful system or library call does **not** set the variable to zero.

If used as a string, yields the corresponding system error string. You can assign a number to `$!` to set *errno* if, for instance, you want "`$!`" to return the string for error *n*, or you want to set the exit value for the `die()` operator. (Mnemonic: What just went bang?)

Also see [Error Indicators](#).

`%!` Each element of `%!` has a true value only if `$!` is set to that value. For example, `${!ENOENT}` is true if and only if the current value of `$!` is `ENOENT`; that is, if the most recent error was "No such file or directory" (or its moral equivalent: not all operating systems give that exact error, and certainly not all languages). To check if a particular key is meaningful on your system, use `exists ${!the_key}`; for a list of legal keys, use `keys %!`. See [Errno](#) for more information, and also see above for the validity of `$!`.

`$EXTENDED_OS_ERROR`

`^E` Error information specific to the current operating system. At the moment, this differs from `$!` under only VMS, OS/2, and Win32 (and for MacPerl). On all other platforms, `^E` is always just the same as `$!`.

Under VMS, `^E` provides the VMS status value from the last system error. This is more specific information about the last system error than that provided by `$!`. This is particularly important when `$!` is set to **EVMSError**.

Under OS/2, `$^E` is set to the error code of the last call to OS/2 API either via CRT, or directly from perl.

Under Win32, `$^E` always returns the last error information reported by the Win32 call `GetLastError()` which describes the last error from within the Win32 API. Most Win32-specific code will report errors via `$^E`. ANSI C and Unix-like calls set `errno` and so most portable Perl code will report errors via `$!`.

Caveats mentioned in the description of `$!` generally apply to `$^E`, also. (Mnemonic: Extra error explanation.)

Also see [Error Indicators](#).

`$EVAL_ERROR`

`$@` The Perl syntax error message from the last `eval()` operator. If `$@` is the null string, the last `eval()` parsed and executed correctly (although the operations you invoked may have failed in the normal fashion). (Mnemonic: Where was the syntax error "at"?)

Warning messages are not collected in this variable. You can, however, set up a routine to process warnings by setting `$SIG{__WARN__}` as described below.

Also see [Error Indicators](#).

`$PROCESS_ID`

`$PID`

`$$` The process number of the Perl running this script. You should consider this variable read-only, although it will be altered across `fork()` calls. (Mnemonic: same as shells.)

`$REAL_USER_ID`

`$UID`

`$<` The real uid of this process. (Mnemonic: it's the uid you came *from*, if you're running `setuid`.) You can change both the real uid and the effective uid at the same time by using `POSIX::setuid()`.

`$EFFECTIVE_USER_ID`

`$EUID`

`$` The effective uid of this process. Example:

```
$< = $>;           # set real to effective uid
($<,$>) = ($>,$<); # swap real and effective uid
```

You can change both the effective uid and the real uid at the same time by using `POSIX::setuid()`.

(Mnemonic: it's the uid you went *to*, if you're running `setuid`.) `$<` and `< $` can be swapped only on machines supporting `setreuid()`.

`$REAL_GROUP_ID`

`$GID`

`$()` The real gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by `getgid()`, and the subsequent ones by `getgroups()`, one of which may be the same as the first number.

However, a value assigned to `$()` must be a single number used to set the real gid. So the value given by `$()` should *not* be assigned back to `$()` without being forced numeric, such as by adding zero.

You can change both the real gid and the effective gid at the same time by using `POSIX::setgid()`.

(Mnemonic: parentheses are used to *group* things. The real gid is the group you *left*, if you're

running `setgid()`.)

`$EFFECTIVE_GROUP_ID`

`$EGID`

`$)` The effective gid of this process. If you are on a machine that supports membership in multiple groups simultaneously, gives a space separated list of groups you are in. The first number is the one returned by `getegid()`, and the subsequent ones by `getgroups()`, one of which may be the same as the first number.

Similarly, a value assigned to `$)` must also be a space-separated list of numbers. The first number sets the effective gid, and the rest (if any) are passed to `setgroups()`. To get the effect of an empty list for `setgroups()`, just repeat the new effective gid; that is, to force an effective gid of 5 and an effectively empty `setgroups()` list, say `$) = "5 5"`.

You can change both the effective gid and the real gid at the same time by using `POSIX::setgid()` (use only a single numeric argument).

(Mnemonic: parentheses are used to *group* things. The effective gid is the group that's *right* for you, if you're running `setgid()`.)

`< $< , < $, $(` (and `$)`) can be set only on machines that support the corresponding `set[re][ug]id()` routine. `$(` (and `$)`) can be swapped only on machines supporting `setregid()`.

`$PROGRAM_NAME`

`$0` Contains the name of the program being executed. On some operating systems assigning to `$0` modifies the argument area that the **ps** program sees. This is more useful as a way of indicating the current program state than it is for hiding the program you're running. (Mnemonic: same as **sh** and **ksh**.)

Note for BSD users: setting `$0` does not completely remove "perl" from the `ps(1)` output. For example, setting `$0` to "foobar" will result in "perl: foobar (perl)". This is an operating system feature.

`$[` The index of the first element in an array, and of the first character in a substring. Default is 0, but you could theoretically set it to 1 to make Perl behave more like **awk** (or Fortran) when subscripting and when evaluating the `index()` and `substr()` functions. (Mnemonic: `[` begins subscripts.)

As of release 5 of Perl, assignment to `$[` is treated as a compiler directive, and cannot influence the behavior of any other file. Its use is highly discouraged.

`$]` The version + patchlevel / 1000 of the Perl interpreter. This variable can be used to determine whether the Perl interpreter executing a script is in the right range of versions. (Mnemonic: Is this version of perl in the right bracket?) Example:

```
warn "No checksumming!\n" if $] < 3.019;
```

See also the documentation of `use VERSION` and `require VERSION` for a convenient way to fail if the running Perl interpreter is too old.

The floating point representation can sometimes lead to inaccurate numeric comparisons. See `$_^V` for a more modern representation of the Perl version that allows accurate string comparisons.

`$COMPILING`

`$_^C` The current value of the flag associated with the `-c` switch. Mainly of use with `-MO=...` to allow code to alter its behavior when being compiled, such as for example to AUTOLOAD at compile time rather than normal, deferred loading. See [perlcc](#). Setting `$_^C = 1` is similar to calling `B::minus_c`.

\$DEBUGGING

\$^D The current value of the debugging flags. (Mnemonic: value of **-D** switch.)

\$SYSTEM_FD_MAX

\$^F The maximum system file descriptor, ordinarily 2. System file descriptors are passed to `exec()`ed processes, while higher file descriptors are not. Also, during an `open()`, system file descriptors are preserved even if the `open()` fails. (Ordinary file descriptors are closed before the `open()` is attempted.) The close-on-exec status of a file descriptor will be decided according to the value of **\$^F** when the corresponding file, pipe, or socket was opened, not the time of the `exec()`.

\$^H WARNING: This variable is strictly for internal use only. Its availability, behavior, and contents are subject to change without notice.

This variable contains compile-time hints for the Perl interpreter. At the end of compilation of a BLOCK the value of this variable is restored to the value when the interpreter started to compile the BLOCK.

When perl begins to parse any block construct that provides a lexical scope (e.g., eval body, required file, subroutine body, loop body, or conditional block), the existing value of **\$^H** is saved, but its value is left unchanged. When the compilation of the block is completed, it regains the saved value. Between the points where its value is saved and restored, code that executes within BEGIN blocks is free to change the value of **\$^H**.

This behavior provides the semantic of lexical scoping, and is used in, for instance, the `use strict` pragma.

The contents should be an integer; different bits of it are used for different pragmatic flags. Here's an example:

```
sub add_100 { $^H |= 0x100 }
sub foo {
    BEGIN { add_100() }
    bar->baz($boon);
}
```

Consider what happens during execution of the BEGIN block. At this point the BEGIN block has already been compiled, but the body of `foo()` is still being compiled. The new value of **\$^H** will therefore be visible only while the body of `foo()` is being compiled.

Substitution of the above BEGIN block with:

```
BEGIN { require strict; strict->import('vars') }
```

demonstrates how `use strict 'vars'` is implemented. Here's a conditional version of the same lexical pragma:

```
BEGIN { require strict; strict->import('vars') if $condition }
```

%^H WARNING: This variable is strictly for internal use only. Its availability, behavior, and contents are subject to change without notice.

The **%^H** hash provides the same scoping semantic as **\$^H**. This makes it useful for implementation of lexically scoped pragmas.

\$INPLACE_EDIT

\$^I The current value of the inplace-edit extension. Use `undef` to disable inplace editing. (Mnemonic: value of **-i** switch.)

`$^M` By default, running out of memory is an untrappable, fatal error. However, if suitably built, Perl can use the contents of `$^M` as an emergency memory pool after `die()`ing. Suppose that your Perl were compiled with `-DPERL_EMERGENCY_SBRK` and used Perl's `malloc`. Then

```
$^M = 'a' x (1 << 16);
```

would allocate a 64K buffer for use in an emergency. See the *INSTALL* file in the Perl distribution for information on how to enable this option. To discourage casual use of this advanced feature, there is no *English/English* long name for this variable.

`$OSNAME`

`$^O` The name of the operating system under which this copy of Perl was built, as determined during the configuration process. The value is identical to `$Config{'osname'}`. See also *Config* and the `-V` command-line switch documented in *perlrun*.

In Windows platforms, `$^O` is not very helpful: since it is always `MSWin32`, it doesn't tell the difference between 95/98/ME/NT/2000/XP/CE/.NET. Use `Win32::GetOSName()` or `Win32::GetOSVersion()` (see *Win32* and *perlport*) to distinguish between the variants.

`$_{^OPEN}`

An internal variable used by `PerlIO`. A string in two parts, separated by a `\0` byte, the first part describes the input layers, the second part describes the output layers.

`$PERLDB`

`$^P` The internal variable for debugging support. The meanings of the various bits are subject to change, but currently indicate:

- `0x01` Debug subroutine enter/exit.
- `0x02` Line-by-line debugging.
- `0x04` Switch off optimizations.
- `0x08` Preserve more data for future interactive inspections.
- `0x10` Keep info about source lines on which a subroutine is defined.
- `0x20` Start with single-step on.
- `0x40` Use subroutine address instead of name when reporting.
- `0x80` Report `goto &subroutine` as well.
- `0x100` Provide informative "file" names for evals based on the place they were compiled.
- `0x200` Provide informative names to anonymous subroutines based on the place they were compiled.

Some bits may be relevant at compile-time only, some at run-time only. This is a new mechanism and the details may change.

`$LAST_REGEXP_CODE_RESULT`

`$^R` The result of evaluation of the last successful `(?{ code })` regular expression assertion (see *perlre*). May be written to.

`$EXCEPTIONS_BEING_CAUGHT`

`$^S` Current state of the interpreter. Undefined if parsing of the current module/eval is not finished (may happen in `$_SIG{__DIE__}` and `$_SIG{__WARN__}` handlers). True if inside an `eval()`, otherwise false.

`$BASETIME`

`$^T` The time at which the program began running, in seconds since the epoch (beginning of 1970). The values returned by the `-M`, `-A`, and `-C` filetests are based on this value.

`${^TAINT}`

Reflects if taint mode is on or off (i.e. if the program was run with `-T` or not). True for on, false for off.

`$PERL_VERSION`

`^V` The revision, version, and subversion of the Perl interpreter, represented as a string composed of characters with those ordinals. Thus in Perl v5.6.0 it equals `chr(5) . chr(6) . chr(0)` and will return true for `^V eq v5.6.0`. Note that the characters in this string value can potentially be in Unicode range.

This can be used to determine whether the Perl interpreter executing a script is in the right range of versions. (Mnemonic: use `^V` for Version Control.) Example:

```
warn "No \"our\" declarations!\n" if ^V and ^V lt v5.6.0;
```

To convert `^V` into its string representation use `sprintf()`'s `"%vd"` conversion:

```
printf "version is v%vd\n", ^V; # Perl's version
```

See the documentation of `use VERSION` and `require VERSION` for a convenient way to fail if the running Perl interpreter is too old.

See also `$]` for an older representation of the Perl version.

`$WARNING`

`^W` The current value of the warning switch, initially true if `-w` was used, false otherwise, but directly modifiable. (Mnemonic: related to the `-w` switch.) See also [warnings](#).

`${^WARNING_BITS}`

The current set of warning checks enabled by the `use warnings` pragma. See the documentation of `warnings` for more details.

`${^WIDE_SYSTEM_CALLS}`

Global flag that enables system calls made by Perl to use wide character APIs native to the system, if available. This is currently only implemented on the Windows platform.

This can also be enabled from the command line using the `-C` switch.

The initial value is typically for compatibility with Perl versions earlier than 5.6, but may be automatically set to 1 by Perl if the system provides a user-settable default (e.g., `$ENV{LC_CTYPE}`).

The `bytes` pragma always overrides the effect of this flag in the current lexical scope. See [bytes](#).

`$EXECUTABLE_NAME`

`^X` The name used to execute the current copy of Perl, from C's `argv[0]`.

Depending on the host operating system, the value of `^X` may be a relative or absolute pathname of the perl program file, or may be the string used to invoke perl but not the pathname of the perl program file. Also, most operating systems permit invoking programs that are not in the `PATH` environment variable, so there is no guarantee that the value of `^X` is in `PATH`. For VMS, the value may or may not include a version number.

You usually can use the value of `^X` to re-invoke an independent copy of the same perl that is currently running, e.g.,

```
@first_run = `^X -le "print int rand 100 for 1..100"`;
```

But recall that not all operating systems support forking or capturing of the output of commands, so this complex statement may not be portable.

It is not safe to use the value of `^X` as a path name of a file, as some operating systems that have a mandatory suffix on executable files do not require use of the suffix when invoking a command. To convert the value of `^X` to a path name, use the following statements:

```
# Build up a set of file names (not command names).
use Config;
$this_perl = ^X;
if (^O ne 'VMS')
    {$this_perl .= $Config{_exe}
     unless $this_perl =~ m/$Config{_exe}$/i;}
```

Because many operating systems permit anyone with read access to the Perl program file to make a copy of it, patch the copy, and then execute the copy, the security-conscious Perl programmer should take care to invoke the installed copy of perl, not the copy referenced by `^X`. The following statements accomplish this goal, and produce a pathname that can be invoked as a command or referenced as a file.

```
use Config;
$secure_perl_path = $Config{perlpath};
if (^O ne 'VMS')
    {$secure_perl_path .= $Config{_exe}
     unless $secure_perl_path =~ m/$Config{_exe}$/i;}
```

ARGV The special filehandle that iterates over command-line filenames in `@ARGV`. Usually written as the null filehandle in the angle operator `< <`. Note that currently ARGV only has its magical effect within the `< <` operator; elsewhere it is just a plain filehandle corresponding to the last file opened by `< <`. In particular, passing `*ARGV` as a parameter to a function that expects a filehandle may not cause your function to automatically read the contents of all the files in `@ARGV`.

\$ARGV contains the name of the current file when reading from `<`.

@ARGV The array `@ARGV` contains the command-line arguments intended for the script. `$#ARGV` is generally the number of arguments minus one, because `$ARGV[0]` is the first argument, *not* the program's command name itself. See `$0` for the command name.

ARGVOUT

The special filehandle that points to the currently open output file when doing edit-in-place processing with `-i`. Useful when you have to do a lot of inserting and don't want to keep modifying `$_`. See [perlrun](#) for the `-i` switch.

@F The array `@F` contains the fields of each line read in when autosplit mode is turned on. See [perlrun](#) for the `-a` switch. This array is package-specific, and must be declared or given a full package name if not in package main when running under `strict 'vars'`.

@INC The array `@INC` contains the list of places that the `do`, `require`, or `use` constructs look for their library files. It initially consists of the arguments to any `-I` command-line switches, followed by the default Perl library, probably `/usr/local/lib/perl`, followed by `"."`, to represent the current directory. (`"."` will not be appended if taint checks are enabled, either by `-T` or by `-t`.) If you need to modify this at runtime, you should use the `use lib` pragma to get the machine-dependent library properly loaded also:

```
use lib '/mypath/libdir/';
use SomeMod;
```

You can also insert hooks into the file inclusion system by putting Perl code directly into `@INC`. Those hooks may be subroutine references, array references or blessed objects. See [require](#) for details.

`@_` Within a subroutine the array `@_` contains the parameters passed to that subroutine. See [perlsub](#).

`%INC` The hash `%INC` contains entries for each filename included via the `do`, `require`, or `use` operators. The key is the filename you specified (with module names converted to pathnames), and the value is the location of the file found. The `require` operator uses this hash to determine whether a particular file has already been included.

If the file was loaded via a hook (e.g. a subroutine reference, see [require](#) for a description of these hooks), this hook is by default inserted into `%INC` in place of a filename. Note, however, that the hook may have set the `%INC` entry by itself to provide some more specific info.

`%ENV`

`$ENV{expr}`

The hash `%ENV` contains your current environment. Setting a value in `ENV` changes the environment for any child processes you subsequently `fork()` off.

`%SIG`

`$SIG{expr}`

The hash `%SIG` contains signal handlers for signals. For example:

```
sub handler {          # 1st argument is signal name
    my($sig) = @_;
    print "Caught a SIG$sig--shutting down\n";
    close(LOG);
    exit(0);
}

$SIG{'INT'} = \&handler;
$SIG{'QUIT'} = \&handler;
...
$SIG{'INT'} = 'DEFAULT'; # restore default action
$SIG{'QUIT'} = 'IGNORE'; # ignore SIGQUIT
```

Using a value of `'IGNORE'` usually has the effect of ignoring the signal, except for the `CHLD` signal. See [perlipc](#) for more about this special case.

Here are some other examples:

```
$SIG{"PIPE"} = "Plumber"; # assumes main::Plumber (not recommended)
$SIG{"PIPE"} = \&Plumber; # just fine; assume current Plumber
$SIG{"PIPE"} = *Plumber;  # somewhat esoteric
$SIG{"PIPE"} = Plumber(); # oops, what did Plumber() return??
```

Be sure not to use a bareword as the name of a signal handler, lest you inadvertently call it.

If your system has the `sigaction()` function then signal handlers are installed using it. This means you get reliable signal handling. If your system has the `SA_RESTART` flag it is used when signals handlers are installed. This means that system calls for which restarting is supported continue rather than returning when a signal arrives. If you want your system calls to be interrupted by signal delivery then do something like this:

```
use POSIX ':signal_h';

my $alarm = 0;
sigaction(SIGALRM, new POSIX::SigAction sub { $alarm = 1 }
    or die "Error setting SIGALRM handler: $!\n");
```

See [POSIX](#).

Certain internal hooks can be also set using the `%SIG` hash. The routine indicated by

`$SIG{__WARN__}` is called when a warning message is about to be printed. The warning message is passed as the first argument. The presence of a `__WARN__` hook causes the ordinary printing of warnings to `STDERR` to be suppressed. You can use this to save warnings in a variable, or turn warnings into fatal errors, like this:

```
local $SIG{__WARN__} = sub { die $_[0] };
eval $proggie;
```

The routine indicated by `$SIG{__DIE__}` is called when a fatal exception is about to be thrown. The error message is passed as the first argument. When a `__DIE__` hook routine returns, the exception processing continues as it would have in the absence of the hook, unless the hook routine itself exits via a `goto`, a loop exit, or a `die()`. The `__DIE__` handler is explicitly disabled during the call, so that you can die from a `__DIE__` handler. Similarly for `__WARN__`.

Due to an implementation glitch, the `$SIG{__DIE__}` hook is called even inside an `eval()`. Do not use this to rewrite a pending exception in `$@`, or as a bizarre substitute for overriding `CORE::GLOBAL::die()`. This strange action at a distance may be fixed in a future release so that `$SIG{__DIE__}` is only called if your program is about to exit, as was the original intent. Any other use is deprecated.

`__DIE__` / `__WARN__` handlers are very special in one respect: they may be called to report (probable) errors found by the parser. In such a case the parser may be in inconsistent state, so any attempt to evaluate Perl code from such a handler will probably result in a segfault. This means that warnings or errors that result from parsing Perl should be used with extreme caution, like this:

```
require Carp if defined $^S;
Carp::confess("Something wrong") if defined &Carp::confess;
die "Something wrong, but could not load Carp to give backtrace...
    To see backtrace try starting Perl with -MCarp switch";
```

Here the first line will load `Carp` *unless* it is the parser who called the handler. The second line will print backtrace and die if `Carp` was available. The third line will be executed only if `Carp` was not available.

See [die](#), [warn](#), [eval](#), and [warnings](#) for additional information.

Error Indicators

The variables `$@`, `$!`, `$^E`, and `$?` contain information about different types of error conditions that may appear during execution of a Perl program. The variables are shown ordered by the "distance" between the subsystem which reported the error and the Perl process. They correspond to errors detected by the Perl interpreter, C library, operating system, or an external program, respectively.

To illustrate the differences between these variables, consider the following Perl expression, which uses a single-quoted string:

```
eval q{
    open my $pipe, "/cdrom/install |" or die $!;
    my @res = <$pipe>;
    close $pipe or die "bad pipe: $?, $!";
};
```

After execution of this statement all 4 variables may have been set.

`$@` is set if the string to be eval-ed did not compile (this may happen if `open` or `close` were imported with bad prototypes), or if Perl code executed during evaluation `die()`d. In these cases the value of `$@` is the compile error, or the argument to `die` (which will interpolate `$!` and `$?!()`). (See also [Fatal](#), though.)

When the `eval()` expression above is executed, `open()`, `<<PIPE`, and `close` are translated to calls in the C run-time library and thence to the operating system kernel. `$!` is set to the C library's `errno` if one

of these calls fails.

Under a few operating systems, `$^E` may contain a more verbose error indicator, such as in this case, "CDROM tray not closed." Systems that do not support extended error messages leave `$^E` the same as `$!`.

Finally, `$?` may be set to non-0 value if the external program `/cdrom/install` fails. The upper eight bits reflect specific error conditions encountered by the program (the program's `exit()` value). The lower eight bits reflect mode of failure, like signal death and core dump information. See `wait(2)` for details. In contrast to `$!` and `$^E`, which are set only if error condition is detected, the variable `$?` is set on each `wait` or `pipe close`, overwriting the old value. This is more like `$@`, which on every `eval()` is always set on failure and cleared on success.

For more details, see the individual descriptions at `$@`, `$!`, `$^E`, and `$?`.

Technical Note on the Syntax of Variable Names

Variable names in Perl can have several formats. Usually, they must begin with a letter or underscore, in which case they can be arbitrarily long (up to an internal limit of 251 characters) and may contain letters, digits, underscores, or the special sequence `::` or `'`. In this case, the part before the last `::` or `'` is taken to be a *package qualifier*; see [perlmod](#).

Perl variable names may also be a sequence of digits or a single punctuation or control character. These names are all reserved for special uses by Perl; for example, the all-digits names are used to hold data captured by backreferences after a regular expression match. Perl has a special syntax for the single-control-character names: It understands `^X` (caret X) to mean the control-X character. For example, the notation `$^W` (dollar-sign caret W) is the scalar variable whose name is the single character control-W. This is better than typing a literal control-W into your program.

Finally, new in Perl 5.6, Perl variable names may be alphanumeric strings that begin with control characters (or better yet, a caret). These variables must be written in the form `${^FOO}`; the braces are not optional. `${^FOO}` denotes the scalar variable whose name is a control-F followed by two o's. These variables are reserved for future special uses by Perl, except for the ones that begin with `^_` (control-underscore or caret-underscore). No control-character name that begins with `^_` will acquire a special meaning in any future version of Perl; such names may therefore be used safely in programs. `^_` itself, however, is reserved.

Perl identifiers that begin with digits, control characters, or punctuation characters are exempt from the effects of the `package` declaration and are always forced to be in package `main`. A few other names are also exempt:

ENV	STDIN
INC	STDOUT
ARGV	STDERR
ARGVOUT	
SIG	

In particular, the new special `${^_XYZ}` variables are always taken to be in package `main`, regardless of any package declarations presently in scope.

BUGS

Due to an unfortunate accident of Perl's implementation, `use English` imposes a considerable performance penalty on all regular expression matches in a program, regardless of whether they occur in the scope of `use English`. For that reason, saying `use English` in libraries is strongly discouraged. See the `Devel::SawAmpersand` module documentation from CPAN (<http://www.cpan.org/modules/by-module/Devel/>) for more information.

Having to even think about the `$^S` variable in your exception handlers is simply wrong.

`$SIG{__DIE__}` as currently implemented invites grievous and difficult to track down errors. Avoid it and use an `END{}` or `CORE::GLOBAL::die` override instead.

NAME

perlxs – XS language reference manual

DESCRIPTION

Introduction

XS is an interface description file format used to create an extension interface between Perl and C code (or a C library) which one wishes to use with Perl. The XS interface is combined with the library to create a new library which can then be either dynamically loaded or statically linked into perl. The XS interface description is written in the XS language and is the core component of the Perl extension interface.

An **XSUB** forms the basic unit of the XS interface. After compilation by the **xsubpp** compiler, each XSUB amounts to a C function definition which will provide the glue between Perl calling conventions and C calling conventions.

The glue code pulls the arguments from the Perl stack, converts these Perl values to the formats expected by a C function, call this C function, transfers the return values of the C function back to Perl. Return values here may be a conventional C return value or any C function arguments that may serve as output parameters. These return values may be passed back to Perl either by putting them on the Perl stack, or by modifying the arguments supplied from the Perl side.

The above is a somewhat simplified view of what really happens. Since Perl allows more flexible calling conventions than C, XSUBs may do much more in practice, such as checking input parameters for validity, throwing exceptions (or returning undef/empty list) if the return value from the C function indicates failure, calling different C functions based on numbers and types of the arguments, providing an object-oriented interface, etc.

Of course, one could write such glue code directly in C. However, this would be a tedious task, especially if one needs to write glue for multiple C functions, and/or one is not familiar enough with the Perl stack discipline and other such arcana. XS comes to the rescue here: instead of writing this glue C code in long-hand, one can write a more concise short-hand *description* of what should be done by the glue, and let the XS compiler **xsubpp** handle the rest.

The XS language allows one to describe the mapping between how the C routine is used, and how the corresponding Perl routine is used. It also allows creation of Perl routines which are directly translated to C code and which are not related to a pre-existing C function. In cases when the C interface coincides with the Perl interface, the XSUB declaration is almost identical to a declaration of a C function (in K&R style). In such circumstances, there is another tool called **h2xs** that is able to translate an entire C header file into a corresponding XS file that will provide glue to the functions/macros described in the header file.

The XS compiler is called **xsubpp**. This compiler creates the constructs necessary to let an XSUB manipulate Perl values, and creates the glue necessary to let Perl call the XSUB. The compiler uses **typemaps** to determine how to map C function parameters and output values to Perl values and back. The default typemap (which comes with Perl) handles many common C types. A supplementary typemap may also be needed to handle any special structures and types for the library being linked.

A file in XS format starts with a C language section which goes until the first **MODULE =** directive. Other XS directives and XSUB definitions may follow this line. The "language" used in this part of the file is usually referred to as the XS language. **xsubpp** recognizes and skips POD (see [perlpod](#)) in both the C and XS language sections, which allows the XS file to contain embedded documentation.

See [perlxstut](#) for a tutorial on the whole extension creation process.

Note: For some extensions, Dave Beazley's SWIG system may provide a significantly more convenient mechanism for creating the extension glue code. See <http://www.swig.org/> for more information.

On The Road

Many of the examples which follow will concentrate on creating an interface between Perl and the ONC+ RPC bind library functions. The `rpcb_gettime()` function is used to demonstrate many features of the

XS language. This function has two parameters; the first is an input parameter and the second is an output parameter. The function also returns a status value.

```
bool_t rpcb_gettime(const char *host, time_t *timep);
```

From C this function will be called with the following statements.

```
#include <rpc/rpc.h>
bool_t status;
time_t timep;
status = rpcb_gettime( "localhost", &timep );
```

If an XSUB is created to offer a direct translation between this function and Perl, then this XSUB will be used from Perl with the following code. The `$status` and `$timep` variables will contain the output of the function.

```
use RPC;
$status = rpcb_gettime( "localhost", $timep );
```

The following XS file shows an XS subroutine, or XSUB, which demonstrates one possible interface to the `rpcb_gettime()` function. This XSUB represents a direct translation between C and Perl and so preserves the interface even from Perl. This XSUB will be invoked from Perl with the usage shown above. Note that the first three `#include` statements, for `EXTERN.h`, `perl.h`, and `XSUB.h`, will always be present at the beginning of an XS file. This approach and others will be expanded later in this document.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include <rpc/rpc.h>

MODULE = RPC  PACKAGE = RPC

bool_t
rpcb_gettime(host,timep)
    char *host
    time_t &timep
    OUTPUT:
        timep
```

Any extension to Perl, including those containing XSUBs, should have a Perl module to serve as the bootstrap which pulls the extension into Perl. This module will export the extension's functions and variables to the Perl program and will cause the extension's XSUBs to be linked into Perl. The following module will be used for most of the examples in this document and should be used from Perl with the `use` command as shown earlier. Perl modules are explained in more detail later in this document.

```
package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw( rpcb_gettime );

bootstrap RPC;
1;
```

Throughout this document a variety of interfaces to the `rpcb_gettime()` XSUB will be explored. The XSUBs will take their parameters in different orders or will take different numbers of parameters. In each case the XSUB is an abstraction between Perl and the real C `rpcb_gettime()` function, and the XSUB must always ensure that the real `rpcb_gettime()` function is called with the correct parameters. This abstraction will allow the programmer to create a more Perl-like interface to the C function.

The Anatomy of an XSUB

The simplest XSUBs consist of 3 parts: a description of the return value, the name of the XSUB routine and the names of its arguments, and a description of types or formats of the arguments.

The following XSUB allows a Perl program to access a C library function called `sin()`. The XSUB will imitate the C function which takes a single argument and returns a single value.

```
double
sin(x)
double x
```

Optionally, one can merge the description of types and the list of argument names, rewriting this as

```
double
sin(double x)
```

This makes this XSUB look similar to an ANSI C declaration. An optional semicolon is allowed after the argument list, as in

```
double
sin(double x);
```

Parameters with C pointer types can have different semantic: C functions with similar declarations

```
bool string_looks_as_a_number(char *s);
bool make_char_uppercase(char *c);
```

are used in absolutely incompatible manner. Parameters to these functions could be described **xsubpp** like this:

```
char * s
char &c
```

Both these XS declarations correspond to the `char*` C type, but they have different semantics, see *"The & Unary Operator"*.

It is convenient to think that the indirection operator `*` should be considered as a part of the type and the address operator `&` should be considered part of the variable. See *"The Typemap"* for more info about handling qualifiers and unary operators in C types.

The function name and the return type must be placed on separate lines and should be flush left-adjusted.

INCORRECT	CORRECT
double sin(x)	double
double x	sin(x)
	double x

The rest of the function description may be indented or left-adjusted. The following example shows a function with its body left-adjusted. Most examples in this document will indent the body for better readability.

```
CORRECT

double
sin(x)
double x
```

More complicated XSUBs may contain many other sections. Each section of an XSUB starts with the corresponding keyword, such as `INIT:` or `CLEANUP:`. However, the first two lines of an XSUB always contain the same data: descriptions of the return type and the names of the function and its parameters. Whatever immediately follows these is considered to be an `INPUT:` section unless explicitly marked with another keyword. (See *The INPUT: Keyword*.)

An XSUB section continues until another section–start keyword is found.

The Argument Stack

The Perl argument stack is used to store the values which are sent as parameters to the XSUB and to store the XSUB's return value(s). In reality all Perl functions (including non-XSUB ones) keep their values on this stack all the same time, each limited to its own range of positions on the stack. In this document the first position on that stack which belongs to the active function will be referred to as position 0 for that function.

XSUBs refer to their stack arguments with the macro **ST(x)**, where *x* refers to a position in this XSUB's part of the stack. Position 0 for that function would be known to the XSUB as ST(0). The XSUB's incoming parameters and outgoing return values always begin at ST(0). For many simple cases the **xsubpp** compiler will generate the code necessary to handle the argument stack by embedding code fragments found in the `typemaps`. In more complex cases the programmer must supply the code.

The RETVAL Variable

The RETVAL variable is a special C variable that is declared automatically for you. The C type of RETVAL matches the return type of the C library function. The **xsubpp** compiler will declare this variable in each XSUB with non-void return type. By default the generated C function will use RETVAL to hold the return value of the C library function being called. In simple cases the value of RETVAL will be placed in ST(0) of the argument stack where it can be received by Perl as the return value of the XSUB.

If the XSUB has a return type of `void` then the compiler will not declare a RETVAL variable for that function. When using a PPCODE: section no manipulation of the RETVAL variable is required, the section may use direct stack manipulation to place output values on the stack.

If PPCODE: directive is not used, void return value should be used only for subroutines which do not return a value, *even if* CODE: directive is used which sets ST(0) explicitly.

Older versions of this document recommended to use void return value in such cases. It was discovered that this could lead to segfaults in cases when XSUB was *truly* void. This practice is now deprecated, and may be not supported at some future version. Use the return value `SV *` in such cases. (Currently **xsubpp** contains some heuristic code which tries to disambiguate between "truly-void" and "old-practice-declared-as-void" functions. Hence your code is at mercy of this heuristics unless you use `SV *` as return value.)

The MODULE Keyword

The MODULE keyword is used to start the XS code and to specify the package of the functions which are being defined. All text preceding the first MODULE keyword is considered C code and is passed through to the output with POD stripped, but otherwise untouched. Every XS module will have a bootstrap function which is used to hook the XSUBs into Perl. The package name of this bootstrap function will match the value of the last MODULE statement in the XS source files. The value of MODULE should always remain constant within the same XS file, though this is not required.

The following example will start the XS code and will place all functions in a package named RPC.

```
MODULE = RPC
```

The PACKAGE Keyword

When functions within an XS source file must be separated into packages the PACKAGE keyword should be used. This keyword is used with the MODULE keyword and must follow immediately after it when used.

```
MODULE = RPC  PACKAGE = RPC
[ XS code in package RPC ]

MODULE = RPC  PACKAGE = RPCB
[ XS code in package RPCB ]

MODULE = RPC  PACKAGE = RPC
```

```
[ XS code in package RPC ]
```

The same package name can be used more than once, allowing for non-contiguous code. This is useful if you have a stronger ordering principle than package names.

Although this keyword is optional and in some cases provides redundant information it should always be used. This keyword will ensure that the XSUBs appear in the desired package.

The PREFIX Keyword

The PREFIX keyword designates prefixes which should be removed from the Perl function names. If the C function is `rpcb_gettime()` and the PREFIX value is `rpcb_` then Perl will see this function as `_gettime()`.

This keyword should follow the PACKAGE keyword when used. If PACKAGE is not used then PREFIX should follow the MODULE keyword.

```
MODULE = RPC    PREFIX = rpcb_
MODULE = RPC    PACKAGE = RPCB    PREFIX = rpcb_
```

The OUTPUT: Keyword

The OUTPUT: keyword indicates that certain function parameters should be updated (new values made visible to Perl) when the XSUB terminates or that certain values should be returned to the calling Perl function. For simple functions which have no CODE: or PPCODE: section, such as the `sin()` function above, the RETVAL variable is automatically designated as an output value. For more complex functions the **xsubpp** compiler will need help to determine which variables are output variables.

This keyword will normally be used to complement the CODE: keyword. The RETVAL variable is not recognized as an output variable when the CODE: keyword is present. The OUTPUT: keyword is used in this situation to tell the compiler that RETVAL really is an output variable.

The OUTPUT: keyword can also be used to indicate that function parameters are output variables. This may be necessary when a parameter has been modified within the function and the programmer would like the update to be seen by Perl.

```
bool_t
rpcb_gettime(host,timep)
    char *host
    time_t &timep
OUTPUT:
    timep
```

The OUTPUT: keyword will also allow an output parameter to be mapped to a matching piece of code rather than to a typemap.

```
bool_t
rpcb_gettime(host,timep)
    char *host
    time_t &timep
OUTPUT:
    timep sv_setnv(ST(1), (double)timep);
```

xsubpp emits an automatic `SvSETMAGIC()` for all parameters in the OUTPUT section of the XSUB, except RETVAL. This is the usually desired behavior, as it takes care of properly invoking 'set' magic on output parameters (needed for hash or array element parameters that must be created if they didn't exist). If for some reason, this behavior is not desired, the OUTPUT section may contain a `SETMAGIC: DISABLE` line to disable it for the remainder of the parameters in the OUTPUT section. Likewise, `SETMAGIC: ENABLE` can be used to reenable it for the remainder of the OUTPUT section. See [perlguts](#) for more details about 'set' magic.

The NO_OUTPUT Keyword

The NO_OUTPUT can be placed as the first token of the XSUB. This keyword indicates that while the C subroutine we provide an interface to has a non-void return type, the return value of this C subroutine should not be returned from the generated Perl subroutine.

With this keyword present *The RETVAL Variable* is created, and in the generated call to the subroutine this variable is assigned to, but the value of this variable is not going to be used in the auto-generated code.

This keyword makes sense only if RETVAL is going to be accessed by the user-supplied code. It is especially useful to make a function interface more Perl-like, especially when the C return value is just an error condition indicator. For example,

```
NO_OUTPUT int
delete_file(char *name)
    POSTCALL:
        if (RETVAL != 0)
            croak("Error %d while deleting file '%s'", RETVAL, name);
```

Here the generated XS function returns nothing on success, and will die() with a meaningful error message on error.

The CODE: Keyword

This keyword is used in more complicated XSUBs which require special handling for the C function. The RETVAL variable is still declared, but it will not be returned unless it is specified in the OUTPUT: section.

The following XSUB is for a C function which requires special handling of its parameters. The Perl usage is given first.

```
$status = rpcb_gettime( "localhost", $timep );
```

The XSUB follows.

```
bool_t
rpcb_gettime(host,timep)
    char *host
    time_t timep
    CODE:
        RETVAL = rpcb_gettime( host, &timep );
    OUTPUT:
        timep
        RETVAL
```

The INIT: Keyword

The INIT: keyword allows initialization to be inserted into the XSUB before the compiler generates the call to the C function. Unlike the CODE: keyword above, this keyword does not affect the way the compiler handles RETVAL.

```
bool_t
rpcb_gettime(host,timep)
    char *host
    time_t &timep
    INIT:
        printf("# Host is %s\n", host );
    OUTPUT:
        timep
```

Another use for the INIT: section is to check for preconditions before making a call to the C function:

```
long long
lldiv(a,b)
```

```

        long long a
        long long b
    INIT:
        if (a == 0 && b == 0)
            XSRETURN_UNDEF;
        if (b == 0)
            croak("lldiv: cannot divide by 0");

```

The NO_INIT Keyword

The NO_INIT keyword is used to indicate that a function parameter is being used only as an output value. The **xsubpp** compiler will normally generate code to read the values of all function parameters from the argument stack and assign them to C variables upon entry to the function. NO_INIT will tell the compiler that some parameters will be used for output rather than for input and that they will be handled before the function terminates.

The following example shows a variation of the `rpcb_gettime()` function. This function uses the `timep` variable only as an output variable and does not care about its initial contents.

```

    bool_t
    rpcb_gettime(host,timep)
        char *host
        time_t &timep = NO_INIT
    OUTPUT:
        timep

```

Initializing Function Parameters

C function parameters are normally initialized with their values from the argument stack (which in turn contains the parameters that were passed to the XSUB from Perl). The typemaps contain the code segments which are used to translate the Perl values to the C parameters. The programmer, however, is allowed to override the typemaps and supply alternate (or additional) initialization code. Initialization code starts with the first `=`, `;` or `+` on a line in the INPUT: section. The only exception happens if this `;` terminates the line, then this `;` is quietly ignored.

The following code demonstrates how to supply initialization code for function parameters. The initialization code is eval'd within double quotes by the compiler before it is added to the output so anything which should be interpreted literally [mainly `$`, `@`, or `\\`] must be protected with backslashes. The variables `$var`, `$arg`, and `$type` can be used as in typemaps.

```

    bool_t
    rpcb_gettime(host,timep)
        char *host = (char *)SvPV($arg,PL_na);
        time_t &timep = 0;
    OUTPUT:
        timep

```

This should not be used to supply default values for parameters. One would normally use this when a function parameter must be processed by another library function before it can be used. Default parameters are covered in the next section.

If the initialization begins with `=`, then it is output in the declaration for the input variable, replacing the initialization supplied by the typemap. If the initialization begins with `;` or `+`, then it is performed after all of the input variables have been declared. In the `;` case the initialization normally supplied by the typemap is not performed. For the `+` case, the declaration for the variable will include the initialization from the typemap. A global variable, `%v`, is available for the truly rare case where information from one initialization is needed in another initialization.

Here's a truly obscure example:

```

    bool_t

```

```

rpcb_gettime(host,timep)
    time_t &timep ; /* \${timep}=@{ [\${timep}=$arg] } */
    char *host + SvOK(\${timep}) ? SvPV($arg,PL_na) : NULL;
OUTPUT:
    timep

```

The construct `\${timep}=@{ [\${timep}=$arg] }` used in the above example has a two-fold purpose: first, when this line is processed by **xsubpp**, the Perl snippet `\${timep}=$arg` is evaluated. Second, the text of the evaluated snippet is output into the generated C file (inside a C comment)! During the processing of `char *host` line, `$arg` will evaluate to `ST(0)`, and `\${timep}` will evaluate to `ST(1)`.

Default Parameter Values

Default values for XSUB arguments can be specified by placing an assignment statement in the parameter list. The default value may be a number, a string or the special string `NO_INIT`. Defaults should always be used on the right-most parameters only.

To allow the XSUB for `rpcb_gettime()` to have a default host value the parameters to the XSUB could be rearranged. The XSUB will then call the real `rpcb_gettime()` function with the parameters in the correct order. This XSUB can be called from Perl with either of the following statements:

```

$status = rpcb_gettime( $timep, $host );

$status = rpcb_gettime( $timep );

```

The XSUB will look like the code which follows. A `CODE:` block is used to call the real `rpcb_gettime()` function with the parameters in the correct order for that function.

```

bool_t
rpcb_gettime(timep,host="localhost")
    char *host
    time_t timep = NO_INIT
CODE:
    RETVAL = rpcb_gettime( host, &timep );
OUTPUT:
    timep
    RETVAL

```

The PREINIT: Keyword

The `PREINIT:` keyword allows extra variables to be declared immediately before or after the declarations of the parameters from the `INPUT:` section are emitted.

If a variable is declared inside a `CODE:` section it will follow any `typemap` code that is emitted for the input parameters. This may result in the declaration ending up after C code, which is C syntax error. Similar errors may happen with an explicit `;-type` or `+-type` initialization of parameters is used (see ["Initializing Function Parameters"](#)). Declaring these variables in an `INIT:` section will not help.

In such cases, to force an additional variable to be declared together with declarations of other variables, place the declaration into a `PREINIT:` section. The `PREINIT:` keyword may be used one or more times within an XSUB.

The following examples are equivalent, but if the code is using complex `typemaps` then the first example is safer.

```

bool_t
rpcb_gettime(timep)
    time_t timep = NO_INIT
PREINIT:
    char *host = "localhost";
CODE:

```

```

        RETVAL = rpcb_gettime( host, &timep );
    OUTPUT:
        timep
    RETVAL

```

For this particular case an INIT: keyword would generate the same C code as the PREINIT: keyword. Another correct, but error-prone example:

```

    bool_t
    rpcb_gettime(timep)
        time_t timep = NO_INIT
    CODE:
        char *host = "localhost";
        RETVAL = rpcb_gettime( host, &timep );
    OUTPUT:
        timep
    RETVAL

```

Another way to declare host is to use a C block in the CODE: section:

```

    bool_t
    rpcb_gettime(timep)
        time_t timep = NO_INIT
    CODE:
        {
            char *host = "localhost";
            RETVAL = rpcb_gettime( host, &timep );
        }
    OUTPUT:
        timep
    RETVAL

```

The ability to put additional declarations before the typemap entries are processed is very handy in the cases when typemap conversions manipulate some global state:

```

MyObject
mutate(o)
    PREINIT:
        MyState st = global_state;
    INPUT:
        MyObject o;
    CLEANUP:
        reset_to(global_state, st);

```

Here we suppose that conversion to MyObject in the INPUT: section and from MyObject when processing RETVAL will modify a global variable global_state. After these conversions are performed, we restore the old value of global_state (to avoid memory leaks, for example).

There is another way to trade clarity for compactness: INPUT sections allow declaration of C variables which do not appear in the parameter list of a subroutine. Thus the above code for mutate() can be rewritten as

```

MyObject
mutate(o)
    MyState st = global_state;
    MyObject o;
    CLEANUP:
        reset_to(global_state, st);

```

and the code for `rpcb_gettime()` can be rewritten as

```
bool_t
rpcb_gettime(timep)
    time_t timep = NO_INIT
    char *host = "localhost";
C_ARGS:
    host, &timep
OUTPUT:
    timep
RETVAL
```

The SCOPE: Keyword

The `SCOPE:` keyword allows scoping to be enabled for a particular XSUB. If enabled, the XSUB will invoke `ENTER` and `LEAVE` automatically.

To support potentially complex type mappings, if a typemap entry used by an XSUB contains a comment like `/*scope*/` then scoping will be automatically enabled for that XSUB.

To enable scoping:

```
SCOPE: ENABLE
```

To disable scoping:

```
SCOPE: DISABLE
```

The INPUT: Keyword

The XSUB's parameters are usually evaluated immediately after entering the XSUB. The `INPUT:` keyword can be used to force those parameters to be evaluated a little later. The `INPUT:` keyword can be used multiple times within an XSUB and can be used to list one or more input variables. This keyword is used with the `PREINIT:` keyword.

The following example shows how the input parameter `timep` can be evaluated late, after a `PREINIT`.

```
bool_t
rpcb_gettime(host,timep)
    char *host
PREINIT:
    time_t tt;
INPUT:
    time_t timep
CODE:
    RETVAL = rpcb_gettime( host, &tt );
    timep = tt;
OUTPUT:
    timep
RETVAL
```

The next example shows each input parameter evaluated late.

```
bool_t
rpcb_gettime(host,timep)
PREINIT:
    time_t tt;
INPUT:
    char *host
PREINIT:
    char *h;
INPUT:
```

```

        time_t timep
CODE:
        h = host;
        RETVAL = rpcb_gettime( h, &tt );
        timep = tt;
OUTPUT:
        timep
        RETVAL

```

Since INPUT sections allow declaration of C variables which do not appear in the parameter list of a subroutine, this may be shortened to:

```

bool_t
rpcb_gettime(host,timep)
        time_t tt;
        char *host;
        char *h = host;
        time_t timep;
CODE:
        RETVAL = rpcb_gettime( h, &tt );
        timep = tt;
OUTPUT:
        timep
        RETVAL

```

(We used our knowledge that input conversion for `char *` is a "simple" one, thus `host` is initialized on the declaration line, and our assignment `h = host` is not performed too early. Otherwise one would need to have the assignment `h = host` in a CODE: or INIT: section.)

The IN/OUTLIST/IN_OUTLIST/OUT/IN_OUT Keywords

In the list of parameters for an XSUB, one can precede parameter names by the IN/OUTLIST/IN_OUTLIST/OUT/IN_OUT keywords. IN keyword is the default, the other keywords indicate how the Perl interface should differ from the C interface.

Parameters preceded by OUTLIST/IN_OUTLIST/OUT/IN_OUT keywords are considered to be used by the C subroutine *via pointers*. OUTLIST/OUT keywords indicate that the C subroutine does not inspect the memory pointed by this parameter, but will write through this pointer to provide additional return values.

Parameters preceded by OUTLIST keyword do not appear in the usage signature of the generated Perl function.

Parameters preceded by IN_OUTLIST/IN_OUT/OUT *do* appear as parameters to the Perl function. With the exception of OUT-parameters, these parameters are converted to the corresponding C type, then pointers to these data are given as arguments to the C function. It is expected that the C function will write through these pointers.

The return list of the generated Perl function consists of the C return value from the function (unless the XSUB is of void return type or The NO_OUTPUT Keyword was used) followed by all the OUTLIST and IN_OUTLIST parameters (in the order of appearance). On the return from the XSUB the IN_OUT/OUT Perl parameter will be modified to have the values written by the C function.

For example, an XSUB

```

void
day_month(OUTLIST day, IN unix_time, OUTLIST month)
        int day
        int unix_time
        int month

```


should be used from Perl as

```
my ($day, $month) = day_month(time);
```

The C signature of the corresponding function should be

```
void day_month(int *day, int unix_time, int *month);
```

The IN/OUTLIST/IN_OUTLIST/IN_OUT/OUT keywords can be mixed with ANSI-style declarations, as in

```
void
day_month(OUTLIST int day, int unix_time, OUTLIST int month)
```

(here the optional IN keyword is omitted).

The IN_OUT parameters are identical with parameters introduced with [The & Unary Operator](#) and put into the OUTPUT: section (see [The OUTPUT: Keyword](#)). The IN_OUTLIST parameters are very similar, the only difference being that the value C function writes through the pointer would not modify the Perl parameter, but is put in the output list.

The OUTLIST/OUT parameter differ from IN_OUTLIST/IN_OUT parameters only by the initial value of the Perl parameter not being read (and not being given to the C function – which gets some garbage instead). For example, the same C function as above can be interfaced with as

```
void day_month(OUT int day, int unix_time, OUT int month);
```

or

```
void
day_month(day, unix_time, month)
    int &day = NO_INIT
    int  unix_time
    int &month = NO_INIT
OUTPUT:
    day
    month
```

However, the generated Perl function is called in very C-ish style:

```
my ($day, $month);
day_month($day, time, $month);
```

The length(NAME) Keyword

If one of the input arguments to the C function is the length of a string argument NAME, one can substitute the name of the length-argument by length(NAME) in the XSUB declaration. This argument must be omitted when the generated Perl function is called. E.g.,

```
void
dump_chars(char *s, short l)
{
    short n = 0;
    while (n < l) {
        printf("s[%d] = \"%\\%#03o\\\"\\n", n, (int)s[n]);
        n++;
    }
}

MODULE = x          PACKAGE = x

void dump_chars(char *s, short length(s))
```

should be called as dump_chars(\$string).

This directive is supported with ANSI-type function declarations only.

Variable-length Parameter Lists

XSUBs can have variable-length parameter lists by specifying an ellipsis (. . .) in the parameter list. This use of the ellipsis is similar to that found in ANSI C. The programmer is able to determine the number of arguments passed to the XSUB by examining the `items` variable which the **xsubpp** compiler supplies for all XSUBs. By using this mechanism one can create an XSUB which accepts a list of parameters of unknown length.

The `host` parameter for the `rpcb_gettime()` XSUB can be optional so the ellipsis can be used to indicate that the XSUB will take a variable number of parameters. Perl should be able to call this XSUB with either of the following statements.

```
$status = rpcb_gettime( $timep, $host );
$status = rpcb_gettime( $timep );
```

The XS code, with ellipsis, follows.

```
bool_t
rpcb_gettime(timep, ...)
    time_t timep = NO_INIT
PREINIT:
    char *host = "localhost";
    STRLEN n_a;
CODE:
    if( items > 1 )
        host = (char *)SvPV(ST(1), n_a);
    RETVAL = rpcb_gettime( host, &timep );
OUTPUT:
    timep
    RETVAL
```

The C_ARGS: Keyword

The `C_ARGS:` keyword allows creating of XSUBS which have different calling sequence from Perl than from C, without a need to write `CODE:` or `PPCODE:` section. The contents of the `C_ARGS:` paragraph is put as the argument to the called C function without any change.

For example, suppose that a C function is declared as

```
symbolic nth_derivative(int n, symbolic function, int flags);
```

and that the default flags are kept in a global C variable `default_flags`. Suppose that you want to create an interface which is called as

```
$second_deriv = $function->nth_derivative(2);
```

To do this, declare the XSUB as

```
symbolic
nth_derivative(function, n)
    symbolic      function
    int           n
C_ARGS:
    n, function, default_flags
```

The PPCODE: Keyword

The `PPCODE:` keyword is an alternate form of the `CODE:` keyword and is used to tell the **xsubpp** compiler that the programmer is supplying the code to control the argument stack for the XSUBs return values. Occasionally one will want an XSUB to return a list of values rather than a single value. In these cases one must use `PPCODE:` and then explicitly push the list of values on the stack. The `PPCODE:` and `CODE:`

keywords should not be used together within the same XSUB.

The actual difference between PPCODE: and CODE: sections is in the initialization of SP macro (which stands for the *current* Perl stack pointer), and in the handling of data on the stack when returning from an XSUB. In CODE: sections SP preserves the value which was on entry to the XSUB: SP is on the function pointer (which follows the last parameter). In PPCODE: sections SP is moved backward to the beginning of the parameter list, which allows PUSH* () macros to place output values in the place Perl expects them to be when the XSUB returns back to Perl.

The generated trailer for a CODE: section ensures that the number of return values Perl will see is either 0 or 1 (depending on the voidness of the return value of the C function, and heuristics mentioned in *"The RETVAL Variable"*). The trailer generated for a PPCODE: section is based on the number of return values and on the number of times SP was updated by [X] PUSH* () macros.

Note that macros ST(i), XST_m* () and XSRETURN* () work equally well in CODE: sections and PPCODE: sections.

The following XSUB will call the C rpcb_gettime () function and will return its two output values, timep and status, to Perl as a single list.

```
void
rpcb_gettime(host)
    char *host
PREINIT:
    time_t  timep;
    bool_t  status;
PPCODE:
    status = rpcb_gettime( host, &timep );
    EXTEND(SP, 2);
    PUSHs(sv_2mortal(newSViv(status)));
    PUSHs(sv_2mortal(newSViv(timep)));
```

Notice that the programmer must supply the C code necessary to have the real rpcb_gettime () function called and to have the return values properly placed on the argument stack.

The void return type for this function tells the **xsubpp** compiler that the RETVAL variable is not needed or used and that it should not be created. In most scenarios the void return type should be used with the PPCODE: directive.

The EXTEND () macro is used to make room on the argument stack for 2 return values. The PPCODE: directive causes the **xsubpp** compiler to create a stack pointer available as SP, and it is this pointer which is being used in the EXTEND () macro. The values are then pushed onto the stack with the PUSHs () macro.

Now the rpcb_gettime () function can be used from Perl with the following statement.

```
($status, $timep) = rpcb_gettime("localhost");
```

When handling output parameters with a PPCODE section, be sure to handle 'set' magic properly. See [perlgbuts](#) for details about 'set' magic.

Returning Undef And Empty Lists

Occasionally the programmer will want to return simply undef or an empty list if a function fails rather than a separate status value. The rpcb_gettime () function offers just this situation. If the function succeeds we would like to have it return the time and if it fails we would like to have undef returned. In the following Perl code the value of \$timep will either be undef or it will be a valid time.

```
$timep = rpcb_gettime( "localhost" );
```

The following XSUB uses the SV * return type as a mnemonic only, and uses a CODE: block to indicate to the compiler that the programmer has supplied all the necessary code. The sv_newmortal () call will initialize the return value to undef, making that the default return value.

```
SV *
rpcb_gettime(host)
    char * host
    PREINIT:
        time_t timep;
        bool_t x;
    CODE:
        ST(0) = sv_newmortal();
        if( rpcb_gettime( host, &timep ) )
            sv_setnv( ST(0), (double)timep);
```

The next example demonstrates how one would place an explicit undef in the return value, should the need arise.

```
SV *
rpcb_gettime(host)
    char * host
    PREINIT:
        time_t timep;
        bool_t x;
    CODE:
        ST(0) = sv_newmortal();
        if( rpcb_gettime( host, &timep ) ){
            sv_setnv( ST(0), (double)timep);
        }
        else{
            ST(0) = &PL_sv_undef;
        }
```

To return an empty list one must use a PPCODE: block and then not push return values on the stack.

```
void
rpcb_gettime(host)
    char *host
    PREINIT:
        time_t timep;
    PPCODE:
        if( rpcb_gettime( host, &timep ) )
            PUSHs(sv_2mortal(newSViv(timep)));
        else{
            /* Nothing pushed on stack, so an empty
             * list is implicitly returned. */
        }
```

Some people may be inclined to include an explicit `return` in the above XSUB, rather than letting control fall through to the end. In those situations `XSRETURN_EMPTY` should be used, instead. This will ensure that the XSUB stack is properly adjusted. Consult [API LISTING in *perlguts*](#) for other `XSRETURN` macros.

Since `XSRETURN_*` macros can be used with CODE blocks as well, one can rewrite this example as:

```
int
rpcb_gettime(host)
    char *host
    PREINIT:
        time_t timep;
    CODE:
        RETVAL = rpcb_gettime( host, &timep );
        if (RETVAL == 0)
```

```

        XSRETURN_UNDEF;
    OUTPUT:
        RETVAL

```

In fact, one can put this check into a POSTCALL: section as well. Together with PREINIT: simplifications, this leads to:

```

    int
    rpcb_gettime(host)
        char *host
        time_t timep;
    POSTCALL:
        if (RETVAL == 0)
            XSRETURN_UNDEF;

```

The REQUIRE: Keyword

The REQUIRE: keyword is used to indicate the minimum version of the **xsubpp** compiler needed to compile the XS module. An XS module which contains the following statement will compile with only **xsubpp** version 1.922 or greater:

```
REQUIRE: 1.922
```

The CLEANUP: Keyword

This keyword can be used when an XSUB requires special cleanup procedures before it terminates. When the CLEANUP: keyword is used it must follow any CODE:, PPCODE:, or OUTPUT: blocks which are present in the XSUB. The code specified for the cleanup block will be added as the last statements in the XSUB.

The POSTCALL: Keyword

This keyword can be used when an XSUB requires special procedures executed after the C subroutine call is performed. When the POSTCALL: keyword is used it must precede OUTPUT: and CLEANUP: blocks which are present in the XSUB.

See examples in *"The NO_OUTPUT Keyword"* and *"Returning Undef And Empty Lists"*.

The POSTCALL: block does not make a lot of sense when the C subroutine call is supplied by user by providing either CODE: or PPCODE: section.

The BOOT: Keyword

The BOOT: keyword is used to add code to the extension's bootstrap function. The bootstrap function is generated by the **xsubpp** compiler and normally holds the statements necessary to register any XSUBs with Perl. With the BOOT: keyword the programmer can tell the compiler to add extra statements to the bootstrap function.

This keyword may be used any time after the first MODULE keyword and should appear on a line by itself. The first blank line after the keyword will terminate the code block.

```

BOOT:
# The following message will be printed when the
# bootstrap function executes.
printf("Hello from the bootstrap!\n");

```

The VERSIONCHECK: Keyword

The VERSIONCHECK: keyword corresponds to **xsubpp**'s `-versioncheck` and `-noverversioncheck` options. This keyword overrides the command line options. Version checking is enabled by default. When version checking is enabled the XS module will attempt to verify that its version matches the version of the PM module.

To enable version checking:

```
VERSIONCHECK: ENABLE
```

To disable version checking:

```
VERSIONCHECK: DISABLE
```

The PROTOTYPES: Keyword

The PROTOTYPES: keyword corresponds to **xsubpp**'s `-prototypes` and `-noprotoypes` options. This keyword overrides the command line options. Prototypes are enabled by default. When prototypes are enabled XSUBs will be given Perl prototypes. This keyword may be used multiple times in an XS module to enable and disable prototypes for different parts of the module.

To enable prototypes:

```
PROTOTYPES: ENABLE
```

To disable prototypes:

```
PROTOTYPES: DISABLE
```

The PROTOTYPE: Keyword

This keyword is similar to the PROTOTYPES: keyword above but can be used to force **xsubpp** to use a specific prototype for the XSUB. This keyword overrides all other prototype options and keywords but affects only the current XSUB. Consult [Prototypes](#) for information about Perl prototypes.

```
bool_t
rpcb_gettime(timep, ...)
    time_t timep = NO_INIT
    PROTOTYPE: $;$
    PREINIT:
        char *host = "localhost";
        STRLEN n_a;
    CODE:
        if( items > 1 )
            host = (char *)SvPV(ST(1), n_a);
        RETVAL = rpcb_gettime( host, &timep );
    OUTPUT:
        timep
        RETVAL
```

If the prototypes are enabled, you can disable it locally for a given XSUB as in the following example:

```
void
rpcb_gettime_noproto()
    PROTOTYPE: DISABLE
...
```

The ALIAS: Keyword

The ALIAS: keyword allows an XSUB to have two or more unique Perl names and to know which of those names was used when it was invoked. The Perl names may be fully-qualified with package names. Each alias is given an index. The compiler will setup a variable called `ix` which contain the index of the alias which was used. When the XSUB is called with its declared name `ix` will be 0.

The following example will create aliases `FOO::_gettime()` and `BAR::getit()` for this function.

```
bool_t
rpcb_gettime(host,timep)
    char *host
    time_t &timep
    ALIAS:
        FOO::_gettime = 1
```

```

        BAR::getit = 2
INIT:
    printf("# ix = %d\n", ix );
OUTPUT:
    timep

```

The OVERLOAD: Keyword

Instead of writing an overloaded interface using pure Perl, you can also use the OVERLOAD keyword to define additional Perl names for your functions (like the ALIAS: keyword above). However, the overloaded functions must be defined with three parameters (except for the nomethod() function which needs four parameters). If any function has the OVERLOAD: keyword, several additional lines will be defined in the c file generated by xsubpp in order to register with the overload magic.

Since blessed objects are actually stored as RV's, it is useful to use the typemap features to preprocess parameters and extract the actual SV stored within the blessed RV. See the sample for T_PTROBJ_SPECIAL below.

To use the OVERLOAD: keyword, create an XS function which takes three input parameters (or use the c style '...' definition) like this:

```

SV *
cmp (lobj, robj, swap)
My_Module_obj    lobj
My_Module_obj    robj
IV               swap
OVERLOAD: cmp <=>
{ /* function defined here */}

```

In this case, the function will overload both of the three way comparison operators. For all overload operations using non-alpha characters, you must type the parameter without quoting, seperating multiple overloads with whitespace. Note that "" (the stringify overload) should be entered as \"\" (i.e. escaped).

The INTERFACE: Keyword

This keyword declares the current XSUB as a keeper of the given calling signature. If some text follows this keyword, it is considered as a list of functions which have this signature, and should be attached to the current XSUB.

For example, if you have 4 C functions multiply(), divide(), add(), subtract() all having the signature:

```
symbolic f(symbolic, symbolic);
```

you can make them all to use the same XSUB using this:

```

symbolic
interface_s_ss(arg1, arg2)
    symbolic    arg1
    symbolic    arg2
INTERFACE:
    multiply divide
    add subtract

```

(This is the complete XSUB code for 4 Perl functions!) Four generated Perl function share names with corresponding C functions.

The advantage of this approach comparing to ALIAS: keyword is that there is no need to code a switch statement, each Perl function (which shares the same XSUB) knows which C function it should call. Additionally, one can attach an extra function remainder() at runtime by using

```

CV *mycv = newXSproto("Symbolic::remainder",
                      XS_Symbolic_interface_s_ss, __FILE__, "$$");

```

```
XSINTERFACE_FUNC_SET(mycv, remainder);
```

say, from another XSUB. (This example supposes that there was no `INTERFACE_MACRO:` section, otherwise one needs to use something else instead of `XSINTERFACE_FUNC_SET`, see the next section.)

The `INTERFACE_MACRO:` Keyword

This keyword allows one to define an `INTERFACE` using a different way to extract a function pointer from an XSUB. The text which follows this keyword should give the name of macros which would extract/set a function pointer. The extractor macro is given return type, `CV*`, and `XSANY.any_dptr` for this `CV*`. The setter macro is given `cv`, and the function pointer.

The default value is `XSINTERFACE_FUNC` and `XSINTERFACE_FUNC_SET`. An `INTERFACE` keyword with an empty list of functions can be omitted if `INTERFACE_MACRO` keyword is used.

Suppose that in the previous example functions pointers for `multiply()`, `divide()`, `add()`, `subtract()` are kept in a global C array `fp[]` with offsets being `multiply_off`, `divide_off`, `add_off`, `subtract_off`. Then one can use

```
#define XSINTERFACE_FUNC_BYOFFSET(ret,cv,f) \
    ((XSINTERFACE_CVT(ret,)) fp[CvXSUBANY(cv).any_i32])
#define XSINTERFACE_FUNC_BYOFFSET_set(cv,f) \
    CvXSUBANY(cv).any_i32 = CAT2(f, _off)
```

in C section,

```
symbolic
interface_s_ss(arg1, arg2)
    symbolic      arg1
    symbolic      arg2
INTERFACE_MACRO:
    XSINTERFACE_FUNC_BYOFFSET
    XSINTERFACE_FUNC_BYOFFSET_set
INTERFACE:
    multiply divide
    add subtract
```

in XSUB section.

The `INCLUDE:` Keyword

This keyword can be used to pull other files into the XS module. The other files may have XS code. `INCLUDE:` can also be used to run a command to generate the XS code to be pulled into the module.

The file *Rpcb1.xsh* contains our `rpcb_gettime()` function:

```
bool_t
rpcb_gettime(host,timep)
    char *host
    time_t &timep
OUTPUT:
    timep
```

The XS module can use `INCLUDE:` to pull that file into it.

```
INCLUDE: Rpcb1.xsh
```

If the parameters to the `INCLUDE:` keyword are followed by a pipe (`|`) then the compiler will interpret the parameters as a command.

```
INCLUDE: cat Rpcb1.xsh |
```


The CASE: Keyword

The CASE: keyword allows an XSUB to have multiple distinct parts with each part acting as a virtual XSUB. CASE: is greedy and if it is used then all other XS keywords must be contained within a CASE:. This means nothing may precede the first CASE: in the XSUB and anything following the last CASE: is included in that case.

A CASE: might switch via a parameter of the XSUB, via the ix ALIAS: variable (see *"The ALIAS: Keyword"*), or maybe via the items variable (see *"Variable-length Parameter Lists"*). The last CASE: becomes the **default** case if it is not associated with a conditional. The following example shows CASE switched via ix with a function rpcb_gettime() having an alias x_gettime(). When the function is called as rpcb_gettime() its parameters are the usual (char *host, time_t *timep), but when the function is called as x_gettime() its parameters are reversed, (time_t *timep, char *host).

```

long
rpcb_gettime(a,b)
CASE: ix == 1
  ALIAS:
    x_gettime = 1
  INPUT:
    # 'a' is timep, 'b' is host
    char *b
    time_t a = NO_INIT
  CODE:
    RETVAL = rpcb_gettime( b, &a );
  OUTPUT:
    a
    RETVAL
CASE:
  # 'a' is host, 'b' is timep
  char *a
  time_t &b = NO_INIT
  OUTPUT:
    b
    RETVAL

```

That function can be called with either of the following statements. Note the different argument lists.

```

$status = rpcb_gettime( $host, $timep );
$status = x_gettime( $timep, $host );

```

The & Unary Operator

The & unary operator in the INPUT: section is used to tell **xsubpp** that it should convert a Perl value to/from C using the C type to the left of &, but provide a pointer to this value when the C function is called.

This is useful to avoid a CODE: block for a C function which takes a parameter by reference. Typically, the parameter should be not a pointer type (an int or long but not an int* or long*).

The following XSUB will generate incorrect C code. The **xsubpp** compiler will turn this into code which calls rpcb_gettime() with parameters (char *host, time_t timep), but the real rpcb_gettime() wants the timep parameter to be of type time_t* rather than time_t.

```

bool_t
rpcb_gettime(host,timep)
  char *host
  time_t timep
  OUTPUT:

```

```
timep
```

That problem is corrected by using the `&` operator. The **xsubpp** compiler will now turn this into code which calls `rpcb_gettime()` correctly with parameters `(char *host, time_t *timep)`. It does this by carrying the `&` through, so the function call looks like `rpcb_gettime(host, &timep)`.

```
bool_t
rpcb_gettime(host,timep)
    char *host
    time_t &timep
OUTPUT:
timep
```

Inserting POD, Comments and C Preprocessor Directives

C preprocessor directives are allowed within `BOOT:`, `PREINIT:`, `INIT:`, `CODE:`, `PPCODE:`, `POSTCALL:`, and `CLEANUP:` blocks, as well as outside the functions. Comments are allowed anywhere after the `MODULE` keyword. The compiler will pass the preprocessor directives through untouched and will remove the commented lines. POD documentation is allowed at any point, both in the C and XS language sections. POD must be terminated with a `=cut` command; **xsubpp** will exit with an error if it does not. It is very unlikely that human generated C code will be mistaken for POD, as most indenting styles result in whitespace in front of any line starting with `=`. Machine generated XS files may fall into this trap unless care is taken to ensure that a space breaks the sequence `"\n="`.

Comments can be added to XSUBs by placing a `#` as the first non-whitespace of a line. Care should be taken to avoid making the comment look like a C preprocessor directive, lest it be interpreted as such. The simplest way to prevent this is to put whitespace in front of the `#`.

If you use preprocessor directives to choose one of two versions of a function, use

```
#if ... version1
#else /* ... version2 */
#endif
```

and not

```
#if ... version1
#endif
#if ... version2
#endif
```

because otherwise **xsubpp** will believe that you made a duplicate definition of the function. Also, put a blank line before the `#else/#endif` so it will not be seen as part of the function body.

Using XS With C++

If an XSUB name contains `::`, it is considered to be a C++ method. The generated Perl function will assume that its first argument is an object pointer. The object pointer will be stored in a variable called `THIS`. The object should have been created by C++ with the `new()` function and should be blessed by Perl with the `sv_setref_pv()` macro. The blessing of the object by Perl can be handled by a `typemap`. An example `typemap` is shown at the end of this section.

If the return type of the XSUB includes `static`, the method is considered to be a static method. It will call the C++ function using the `class::method()` syntax. If the method is not static the function will be called using the `THIS->method()` syntax.

The next examples will use the following C++ class.

```
class color {
public:
    color();
    ~color();
    int blue();
```

```

        void set_blue( int );

    private:
        int c_blue;
};

```

The XSUBs for the `blue()` and `set_blue()` methods are defined with the class name but the parameter for the object (THIS, or "self") is implicit and is not listed.

```

int
color::blue()

void
color::set_blue( val )
    int val

```

Both Perl functions will expect an object as the first parameter. In the generated C++ code the object is called THIS, and the method call will be performed on this object. So in the C++ code the `blue()` and `set_blue()` methods will be called as this:

```

RETVAL = THIS->blue();

THIS->set_blue( val );

```

You could also write a single get/set method using an optional argument:

```

int
color::blue( val = NO_INIT )
    int val
    PROTOTYPE $;$
    CODE:
        if (items > 1)
            THIS->set_blue( val );
        RETVAL = THIS->blue();
    OUTPUT:
        RETVAL

```

If the function's name is **DESTROY** then the C++ delete function will be called and THIS will be given as its parameter. The generated C++ code for

```

void
color::DESTROY()

```

will look like this:

```

color *THIS = ...; // Initialized as in typemap

delete THIS;

```

If the function's name is **new** then the C++ new function will be called to create a dynamic C++ object. The XSUB will expect the class name, which will be kept in a variable called CLASS, to be given as the first argument.

```

color *
color::new()

```

The generated C++ code will call new.

```

RETVAL = new color();

```

The following is an example of a typemap that could be used for this C++ example.

```

TYPENAME
color *                O_OBJECT

```

```

OUTPUT
# The Perl object is blessed into 'CLASS', which should be a
# char* having the name of the package for the blessing.
O_OBJECT
    sv_setref_pv( $arg, CLASS, (void*)$var );

INPUT
O_OBJECT
    if( sv_isobject($arg) && (SvTYPE(SvRV($arg)) == SVt_PVMG) )
        $var = ($type)SvIV((SV*)SvRV( $arg ));
    else{
        warn( "\"${Package}::$func_name() -- $var is not a blessed SV reference" );
        XSRETURN_UNDEF;
    }

```

Interface Strategy

When designing an interface between Perl and a C library a straight translation from C to XS (such as created by `h2xs -x`) is often sufficient. However, sometimes the interface will look very C-like and occasionally nonintuitive, especially when the C function modifies one of its parameters, or returns failure inband (as in "negative return values mean failure"). In cases where the programmer wishes to create a more Perl-like interface the following strategy may help to identify the more critical parts of the interface.

Identify the C functions with input/output or output parameters. The XSUBs for these functions may be able to return lists to Perl.

Identify the C functions which use some inband info as an indication of failure. They may be candidates to return undef or an empty list in case of failure. If the failure may be detected without a call to the C function, you may want to use an INIT: section to report the failure. For failures detectable after the C function returns one may want to use a POSTCALL: section to process the failure. In more complicated cases use CODE: or PPCODE: sections.

If many functions use the same failure indication based on the return value, you may want to create a special typedef to handle this situation. Put

```
typedef int negative_is_failure;
```

near the beginning of XS file, and create an OUTPUT typemap entry for `negative_is_failure` which converts negative values to undef, or maybe `croak()`s. After this the return value of type `negative_is_failure` will create more Perl-like interface.

Identify which values are used by only the C and XSUB functions themselves, say, when a parameter to a function should be a contents of a global variable. If Perl does not need to access the contents of the value then it may not be necessary to provide a translation for that value from C to Perl.

Identify the pointers in the C function parameter lists and return values. Some pointers may be used to implement input/output or output parameters, they can be handled in XS with the `&` unary operator, and, possibly, using the `NO_INIT` keyword. Some others will require handling of types like `int *`, and one needs to decide what a useful Perl translation will do in such a case. When the semantic is clear, it is advisable to put the translation into a typemap file.

Identify the structures used by the C functions. In many cases it may be helpful to use the `T_PTROBJ` typemap for these structures so they can be manipulated by Perl as blessed objects. (This is handled automatically by `h2xs -x`.)

If the same C type is used in several different contexts which require different translations, typedef several new types mapped to this C type, and create separate **typemap** entries for these new types. Use these types in declarations of return type and parameters to XSUBs.

Perl Objects And C Structures

When dealing with C structures one should select either **T_PTROBJ** or **T_PTRREF** for the XS type. Both types are designed to handle pointers to complex objects. The **T_PTRREF** type will allow the Perl object to be unblessed while the **T_PTROBJ** type requires that the object be blessed. By using **T_PTROBJ** one can achieve a form of type-checking because the XSUB will attempt to verify that the Perl object is of the expected type.

The following XS code shows the `getnetconfig()` function which is used with **ONC+ TIRPC**. The `getnetconfig()` function will return a pointer to a C structure and has the C prototype shown below. The example will demonstrate how the C pointer will become a Perl reference. Perl will consider this reference to be a pointer to a blessed object and will attempt to call a destructor for the object. A destructor will be provided in the XS source to free the memory used by `getnetconfig()`. Destructors in XS can be created by specifying an XSUB function whose name ends with the word **DESTROY**. XS destructors can be used to free memory which may have been malloc'd by another XSUB.

```
struct netconfig *getnetconfig(const char *netid);
```

A typedef will be created for `struct netconfig`. The Perl object will be blessed in a class matching the name of the C type, with the tag **Ptr** appended, and the name should not have embedded spaces if it will be a Perl package name. The destructor will be placed in a class corresponding to the class of the object and the **PREFIX** keyword will be used to trim the name to the word **DESTROY** as Perl will expect.

```
typedef struct netconfig Netconfig;

MODULE = RPC    PACKAGE = RPC

Netconfig *
getnetconfig(netid)
    char *netid

MODULE = RPC    PACKAGE = NetconfigPtr    PREFIX = rpcb_

void
rpcb_DESTROY(netconf)
    Netconfig *netconf
CODE:
    printf("Now in NetconfigPtr::DESTROY\n");
    free( netconf );
```

This example requires the following typemap entry. Consult the typemap section for more information about adding new typemaps for an extension.

```
TYPEMAP
Netconfig *    T_PTROBJ
```

This example will be used with the following Perl statements.

```
use RPC;
$netconf = getnetconfig("udp");
```

When Perl destroys the object referenced by `$netconf` it will send the object to the supplied XSUB **DESTROY** function. Perl cannot determine, and does not care, that this object is a C struct and not a Perl object. In this sense, there is no difference between the object created by the `getnetconfig()` XSUB and an object created by a normal Perl subroutine.

The Typemap

The typemap is a collection of code fragments which are used by the **xsubpp** compiler to map C function parameters and values to Perl values. The typemap file may consist of three sections labelled **TYPEMAP**, **INPUT**, and **OUTPUT**. An unlabelled initial section is assumed to be a **TYPEMAP** section. The **INPUT** section tells the compiler how to translate Perl values into variables of certain C types. The **OUTPUT** section tells the compiler how to translate the values from certain C types into values Perl can understand.

The TYPEMAP section tells the compiler which of the INPUT and OUTPUT code fragments should be used to map a given C type to a Perl value. The section labels TYPEMAP, INPUT, or OUTPUT must begin in the first column on a line by themselves, and must be in uppercase.

The default typemap in the `lib/ExtUtils` directory of the Perl source contains many useful types which can be used by Perl extensions. Some extensions define additional typemaps which they keep in their own directory. These additional typemaps may reference INPUT and OUTPUT maps in the main typemap. The **xsubpp** compiler will allow the extension's own typemap to override any mappings which are in the default typemap.

Most extensions which require a custom typemap will need only the TYPEMAP section of the typemap file. The custom typemap used in the `getnetconfig()` example shown earlier demonstrates what may be the typical use of extension typemaps. That typemap is used to equate a C structure with the `T_PTROBJ` typemap. The typemap used by `getnetconfig()` is shown here. Note that the C type is separated from the XS type with a tab and that the C unary operator `*` is considered to be a part of the C type name.

```

TYPEMAP
Netconfig *<tab>T_PTROBJ

```

Here's a more complicated example: suppose that you wanted `struct netconfig` to be blessed into the class `Net::Config`. One way to do this is to use underscores (`_`) to separate package names, as follows:

```
typedef struct netconfig * Net_Config;
```

And then provide a typemap entry `T_PTROBJ_SPECIAL` that maps underscores to double-colons (`::`), and declare `Net_Config` to be of that type:

```

TYPEMAP
Net_Config      T_PTROBJ_SPECIAL

INPUT
T_PTROBJ_SPECIAL
    if (sv_derived_from($arg, "\"${(my $ntt=$ntype)=~s/_/:/g;$ntt}\"")) {
        IV tmp = SvIV((SV*)SvRV($arg));
        $var = ($type) tmp;
    }
    else
        croak("\"$var is not of type ${(my $ntt=$ntype)=~s/_/:/g;$ntt}\"");

OUTPUT
T_PTROBJ_SPECIAL
    sv_setref_pv($arg, "\"${(my $ntt=$ntype)=~s/_/:/g;$ntt}\"",
        (void*)$var);

```

The INPUT and OUTPUT sections substitute underscores for double-colons on the fly, giving the desired effect. This example demonstrates some of the power and versatility of the typemap facility.

Safely Storing Static Data in XS

Starting with Perl 5.8, a macro framework has been defined to allow static data to be safely stored in XS modules that will be accessed from a multi-threaded Perl.

Although primarily designed for use with multi-threaded Perl, the macros have been designed so that they will work with non-threaded Perl as well.

It is therefore strongly recommended that these macros be used by all XS modules that make use of static data.

The easiest way to get a template set of macros to use is by specifying the `-g` (`-global`) option with `h2xs` (see [h2xs](#)).

Below is an example module that makes use of the macros.

```

#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

/* Global Data */

#define MY_CXT_KEY "BlindMice::_guts" XS_VERSION

typedef struct {
    int count;
    char name[3][100];
} my_cxt_t;

START_MY_CXT

MODULE = BlindMice          PACKAGE = BlindMice

BOOT:
{
    MY_CXT_INIT;
    MY_CXT.count = 0;
    strcpy(MY_CXT.name[0], "None");
    strcpy(MY_CXT.name[1], "None");
    strcpy(MY_CXT.name[2], "None");
}

int
newMouse(char * name)
    char * name;
    PREINIT:
        dMY_CXT;
    CODE:
        if (MY_CXT.count >= 3) {
            warn("Already have 3 blind mice") ;
            RETVAL = 0;
        }
        else {
            RETVAL = ++ MY_CXT.count;
            strcpy(MY_CXT.name[MY_CXT.count - 1], name);
        }

char *
get_mouse_name(index)
    int index
    CODE:
        dMY_CXT;
        RETVAL = MY_CXT.lives ++;
        if (index > MY_CXT.count)
            croak("There are only 3 blind mice.");
        else
            RETVAL = newSVpv(MY_CXT.name[index - 1]);

```

REFERENCE

MY_CXT_KEY

This macro is used to define a unique key to refer to the static data for an XS module. The suggested naming scheme, as used by h2xs, is to use a string that consists of the module name, the string "::_guts" and the module version number.

```
#define MY_CXT_KEY "MyModule::_guts" XS_VERSION
```

typedef my_cxt_t

This struct typedef *must* always be called `my_cxt_t` — the other CXT* macros assume the existence of the `my_cxt_t` typedef name.

Declare a typedef named `my_cxt_t` that is a structure that contains all the data that needs to be interpreter-local.

```
typedef struct {
    int some_value;
} my_cxt_t;
```

START_MY_CXT

Always place the `START_MY_CXT` macro directly after the declaration of `my_cxt_t`.

MY_CXT_INIT

The `MY_CXT_INIT` macro initialises storage for the `my_cxt_t` struct.

It *must* be called exactly once — typically in a BOOT: section.

dMY_CXT

Use the `dMY_CXT` macro (a declaration) in all the functions that access `MY_CXT`.

MY_CXT

Use the `MY_CXT` macro to access members of the `my_cxt_t` struct. For example, if `my_cxt_t` is

```
typedef struct {
    int index;
} my_cxt_t;
```

then use this to access the `index` member

```
dMY_CXT;
MY_CXT.index = 2;
```

EXAMPLES

File `RPC.xs`: Interface to some ONC+ RPC bind library functions.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include <rpc/rpc.h>

typedef struct netconfig Netconfig;

MODULE = RPC    PACKAGE = RPC

SV *
rpcb_gettime(host="localhost")
    char *host
    PREINIT:
        time_t timep;
    CODE:
        ST(0) = sv_newmortal();
        if( rpcb_gettime( host, &timep ) )
            sv_setnv( ST(0), (double)timep );

Netconfig *
getnetconfignt(netid="udp")
```



```

        char *netid

MODULE = RPC  PACKAGE = NetconfigPtr  PREFIX = rpcb_

void
rpcb_DESTROY(netconf)
    Netconfig *netconf
CODE:
    printf("NetconfigPtr::DESTROY\n");
    free( netconf );

```

File `typemap`: Custom typemap for `RPC.xs`.

```

TYPEMAP
Netconfig *  T_PTROBJ

```

File `RPC.pm`: Perl module for the `RPC` extension.

```

package RPC;

require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw(rpcb_gettime getnetconfigent);

bootstrap RPC;
1;

```

File `rpctest.pl`: Perl test program for the `RPC` extension.

```

use RPC;

$netconf = getnetconfigent();
$a = rpcb_gettime();
print "time = $a\n";
print "netconf = $netconf\n";

$netconf = getnetconfigent("tcp");
$a = rpcb_gettime("poplar");
print "time = $a\n";
print "netconf = $netconf\n";

```

XS VERSION

This document covers features supported by `xsubpp` 1.935.

AUTHOR

Originally written by Dean Roehrich <roehrich@cray.com>.

Maintained since 1996 by The Perl Porters <perlbug@perl.org>.

NAME

perlXSut – Tutorial for writing XSUBs

DESCRIPTION

This tutorial will educate the reader on the steps involved in creating a Perl extension. The reader is assumed to have access to *perlguts*, *perlapi* and *perlx*.

This tutorial starts with very simple examples and becomes more complex, with each new example adding new features. Certain concepts may not be completely explained until later in the tutorial in order to slowly ease the reader into building extensions.

This tutorial was written from a Unix point of view. Where I know them to be otherwise different for other platforms (e.g. Win32), I will list them. If you find something that was missed, please let me know.

SPECIAL NOTES**make**

This tutorial assumes that the make program that Perl is configured to use is called `make`. Instead of running "make" in the examples that follow, you may have to substitute whatever make program Perl has been configured to use. Running **perl -V:make** should tell you what it is.

Version caveat

When writing a Perl extension for general consumption, one should expect that the extension will be used with versions of Perl different from the version available on your machine. Since you are reading this document, the version of Perl on your machine is probably 5.005 or later, but the users of your extension may have more ancient versions.

To understand what kinds of incompatibilities one may expect, and in the rare case that the version of Perl on your machine is older than this document, see the section on "Troubleshooting these Examples" for more information.

If your extension uses some features of Perl which are not available on older releases of Perl, your users would appreciate an early meaningful warning. You would probably put this information into the **README** file, but nowadays installation of extensions may be performed automatically, guided by *CPAN.pm* module or other tools.

In MakeMaker-based installations, *Makefile.PL* provides the earliest opportunity to perform version checks. One can put something like this in *Makefile.PL* for this purpose:

```
eval { require 5.007 }
    or die <<EOD;
#####
### This module uses frobnication framework which is not available before
### version 5.007 of Perl. Upgrade your Perl before installing Kara::Mba.
#####
EOD
```

Dynamic Loading versus Static Loading

It is commonly thought that if a system does not have the capability to dynamically load a library, you cannot build XSUBs. This is incorrect. You *can* build them, but you must link the XSUBs subroutines with the rest of Perl, creating a new executable. This situation is similar to Perl 4.

This tutorial can still be used on such a system. The XSUB build mechanism will check the system and build a dynamically-loadable library if possible, or else a static library and then, optionally, a new statically-linked executable with that static library linked in.

Should you wish to build a statically-linked executable on a system which can dynamically load libraries, you may, in all the following examples, where the command "make" with no arguments is executed, run the command "make perl" instead.

If you have generated such a statically-linked executable by choice, then instead of saying "make test", you should say "make test_static". On systems that cannot build dynamically-loadable libraries at all, simply saying "make test" is sufficient.

TUTORIAL

Now let's go on with the show!

EXAMPLE 1

Our first extension will be very simple. When we call the routine in the extension, it will print out a well-known message and return.

Run "h2xs -A -n Mytest". This creates a directory named Mytest, possibly under ext/ if that directory exists in the current working directory. Several files will be created in the Mytest dir, including MANIFEST, Makefile.PL, Mytest.pm, Mytest.xs, test.pl, and Changes.

The MANIFEST file contains the names of all the files just created in the Mytest directory.

The file Makefile.PL should look something like this:

```
use ExtUtils::MakeMaker;
# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.
WriteMakefile(
    NAME         => 'Mytest',
    VERSION_FROM => 'Mytest.pm', # finds $VERSION
    LIBS         => [],          # e.g., '-lm'
    DEFINE       => '',          # e.g., '-DHAVE_SOMETHING'
    INC          => '',          # e.g., '-I/usr/include/other'
);
```

The file Mytest.pm should start with something like this:

```
package Mytest;

use strict;
use warnings;

require Exporter;
require DynaLoader;

our @ISA = qw(Exporter DynaLoader);
# Items to export into callers namespace by default. Note: do not export
# names by default without a very good reason. Use EXPORT_OK instead.
# Do not simply export all your public functions/methods/constants.
our @EXPORT = qw(
);
our $VERSION = '0.01';

bootstrap Mytest $VERSION;

# Preloaded methods go here.

# Autoload methods go after __END__, and are processed by the autosplit program
1;
__END__
# Below is the stub of documentation for your module. You better edit it!
```

The rest of the .pm file contains sample code for providing documentation for the extension.

Finally, the Mytest.xs file should look something like this:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

MODULE = Mytest          PACKAGE = Mytest
```

Let's edit the .xs file by adding this to the end of the file:

```
void
hello()
    CODE:
        printf("Hello, world!\n");
```

It is okay for the lines starting at the "CODE:" line to not be indented. However, for readability purposes, it is suggested that you indent CODE: one level and the lines following one more level.

Now we'll run "perl Makefile.PL". This will create a real Makefile, which make needs. Its output looks something like:

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Mytest
%
```

Now, running make will produce output that looks something like this (some long lines have been shortened for clarity and some extraneous lines have been deleted):

```
% make
umask 0 && cp Mytest.pm ./blib/Mytest.pm
perl xsubpp -typemap typemap Mytest.xs >Mytest.tc && mv Mytest.tc Mytest.c
Please specify prototyping behavior for Mytest.xs (see perlxs manual)
cc -c Mytest.c
Running Mkbootstrap for Mytest ( )
chmod 644 Mytest.bs
LD_RUN_PATH="" ld -o ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl -b Mytest.o
chmod 755 ./blib/PA-RISC1.1/auto/Mytest/Mytest.sl
cp Mytest.bs ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
chmod 644 ./blib/PA-RISC1.1/auto/Mytest/Mytest.bs
Manifesting ./blib/man3/Mytest.3
%
```

You can safely ignore the line about "prototyping behavior" – it is explained in the section "The PROTOTYPES: Keyword" in [perlx](#).

If you are on a Win32 system, and the build process fails with linker errors for functions in the C library, check if your Perl is configured to use PerlCRT (running **perl -V:libc** should show you if this is the case). If Perl is configured to use PerlCRT, you have to make sure PerlCRT.lib is copied to the same location that msvcr.lib lives in, so that the compiler can find it on its own. msvcr.lib is usually found in the Visual C compiler's lib directory (e.g. C:/DevStudio/VC/lib).

Perl has its own special way of easily writing test scripts, but for this example only, we'll create our own test script. Create a file called hello that looks like this:

```
#!/opt/perl5/bin/perl

use ExtUtils::testlib;

use Mytest;

Mytest::hello();
```

Now we make the script executable (`chmod -x hello`), run the script and we should see the following output:

```
% ./hello
Hello, world!
%
```

EXAMPLE 2

Now let's add to our extension a subroutine that will take a single numeric argument as input and return 0 if the number is even or 1 if the number is odd.

Add the following to the end of `Mytest.xs`:

```
int
is_even(input)
    int      input
CODE:
    RETVAL = (input % 2 == 0);
OUTPUT:
    RETVAL
```

There does not need to be white space at the start of the `"int input"` line, but it is useful for improving readability. Placing a semi-colon at the end of that line is also optional. Any amount and kind of white space may be placed between the `"int"` and `"input"`.

Now re-run `make` to rebuild our new shared library.

Now perform the same steps as before, generating a `Makefile` from the `Makefile.PL` file, and running `make`.

In order to test that our extension works, we now need to look at the file `test.pl`. This file is set up to imitate the same kind of testing structure that Perl itself has. Within the test script, you perform a number of tests to confirm the behavior of the extension, printing "ok" when the test is correct, "not ok" when it is not. Change the print statement in the `BEGIN` block to print `"1..4"`, and add the following code to the end of the file:

```
print &Mytest::is_even(0) == 1 ? "ok 2" : "not ok 2", "\n";
print &Mytest::is_even(1) == 0 ? "ok 3" : "not ok 3", "\n";
print &Mytest::is_even(2) == 1 ? "ok 4" : "not ok 4", "\n";
```

We will be calling the test script through the command `"make test"`. You should see output that looks something like this:

```
% make test
PERL_DL_NONLAZY=1 /opt/perl5.004/bin/perl (lots of -I arguments) test.pl
1..4
ok 1
ok 2
ok 3
ok 4
%
```

What has gone on?

The program `h2xs` is the starting point for creating extensions. In later examples we'll see how we can use `h2xs` to read header files and generate templates to connect to C routines.

`h2xs` creates a number of files in the extension directory. The file `Makefile.PL` is a perl script which will generate a true `Makefile` to build the extension. We'll take a closer look at it later.

The `.pm` and `.xs` files contain the meat of the extension. The `.xs` file holds the C routines that make up the extension. The `.pm` file contains routines that tell Perl how to load your extension.

Generating the `Makefile` and running `make` created a directory called `blib` (which stands for "build library") in the current working directory. This directory will contain the shared library that we will build. Once we

have tested it, we can install it into its final location.

Invoking the test script via "make test" did something very important. It invoked perl with all those `-I` arguments so that it could find the various files that are part of the extension. It is *very* important that while you are still testing extensions that you use "make test". If you try to run the test script all by itself, you will get a fatal error. Another reason it is important to use "make test" to run your test script is that if you are testing an upgrade to an already-existing version, using "make test" insures that you will test your new extension, not the already-existing version.

When Perl sees a `use extension;`, it searches for a file with the same name as the `use`'d extension that has a `.pm` suffix. If that file cannot be found, Perl dies with a fatal error. The default search path is contained in the `@INC` array.

In our case, `Mytest.pm` tells perl that it will need the `Exporter` and `Dynamic Loader` extensions. It then sets the `@ISA` and `@EXPORT` arrays and the `$VERSION` scalar; finally it tells perl to bootstrap the module. Perl will call its dynamic loader routine (if there is one) and load the shared library.

The two arrays `@ISA` and `@EXPORT` are very important. The `@ISA` array contains a list of other packages in which to search for methods (or subroutines) that do not exist in the current package. This is usually only important for object-oriented extensions (which we will talk about much later), and so usually doesn't need to be modified.

The `@EXPORT` array tells Perl which of the extension's variables and subroutines should be placed into the calling package's namespace. Because you don't know if the user has already used your variable and subroutine names, it's vitally important to carefully select what to export. Do *not* export method or variable names *by default* without a good reason.

As a general rule, if the module is trying to be object-oriented then don't export anything. If it's just a collection of functions and variables, then you can export them via another array, called `@EXPORT_OK`. This array does not automatically place its subroutine and variable names into the namespace unless the user specifically requests that this be done.

See [perlmod](#) for more information.

The `$VERSION` variable is used to ensure that the `.pm` file and the shared library are "in sync" with each other. Any time you make changes to the `.pm` or `.xs` files, you should increment the value of this variable.

Writing good test scripts

The importance of writing good test scripts cannot be overemphasized. You should closely follow the "ok/not ok" style that Perl itself uses, so that it is very easy and unambiguous to determine the outcome of each test case. When you find and fix a bug, make sure you add a test case for it.

By running "make test", you ensure that your `test.pl` script runs and uses the correct version of your extension. If you have many test cases, you might want to copy Perl's test style. Create a directory named "t" in the extension's directory and append the suffix ".t" to the names of your test files. When you run "make test", all of these test files will be executed.

EXAMPLE 3

Our third extension will take one argument as its input, round off that value, and set the *argument* to the rounded value.

Add the following to the end of `Mytest.xs`:

```
void
round(arg)
    double  arg
CODE:
    if (arg > 0.0) {
        arg = floor(arg + 0.5);
    } else if (arg < 0.0) {
        arg = ceil(arg - 0.5);
```

```

        } else {
            arg = 0.0;
        }
    OUTPUT:
        arg

```

Edit the Makefile.PL file so that the corresponding line looks like this:

```
'LIBS'      => ['-lm'],    # e.g., '-lm'
```

Generate the Makefile and run make. Change the BEGIN block to print "1.9" and add the following to test.pl:

```

$i = -1.5; &Mytest::round($i); print $i == -2.0 ? "ok 5" : "not ok 5", "\n";
$i = -1.1; &Mytest::round($i); print $i == -1.0 ? "ok 6" : "not ok 6", "\n";
$i = 0.0; &Mytest::round($i); print $i == 0.0 ? "ok 7" : "not ok 7", "\n";
$i = 0.5; &Mytest::round($i); print $i == 1.0 ? "ok 8" : "not ok 8", "\n";
$i = 1.2; &Mytest::round($i); print $i == 1.0 ? "ok 9" : "not ok 9", "\n";

```

Running "make test" should now print out that all nine tests are okay.

Notice that in these new test cases, the argument passed to round was a scalar variable. You might be wondering if you can round a constant or literal. To see what happens, temporarily add the following line to test.pl:

```
&Mytest::round(3);
```

Run "make test" and notice that Perl dies with a fatal error. Perl won't let you change the value of constants!

What's new here?

- We've made some changes to Makefile.PL. In this case, we've specified an extra library to be linked into the extension's shared library, the math library libm in this case. We'll talk later about how to write XSUBs that can call every routine in a library.
- The value of the function is not being passed back as the function's return value, but by changing the value of the variable that was passed into the function. You might have guessed that when you saw that the return value of round is of type "void".

Input and Output Parameters

You specify the parameters that will be passed into the XSUB on the line(s) after you declare the function's return value and name. Each input parameter line starts with optional white space, and may have an optional terminating semicolon.

The list of output parameters occurs at the very end of the function, just before after the OUTPUT: directive. The use of RETVAL tells Perl that you wish to send this value back as the return value of the XSUB function. In Example 3, we wanted the "return value" placed in the original variable which we passed in, so we listed it (and not RETVAL) in the OUTPUT: section.

The XSUBPP Program

The **xsubpp** program takes the XS code in the .xs file and translates it into C code, placing it in a file whose suffix is .c. The C code created makes heavy use of the C functions within Perl.

The TYPemap file

The **xsubpp** program uses rules to convert from Perl's data types (scalar, array, etc.) to C's data types (int, char, etc.). These rules are stored in the typemap file (\$PERLLIB/ExtUtils/typemap). This file is split into three parts.

The first section maps various C data types to a name, which corresponds somewhat with the various Perl types. The second section contains C code which **xsubpp** uses to handle input parameters. The third section contains C code which **xsubpp** uses to handle output parameters.

Let's take a look at a portion of the .c file created for our extension. The file name is Mytest.c:

```
XS(XS_Mytest_round)
{
    dXSARGS;
    if (items != 1)
        croak("Usage: Mytest::round(arg)");
    {
        double arg = (double)SvNV(ST(0));    /* XXXXXX */
        if (arg > 0.0) {
            arg = floor(arg + 0.5);
        } else if (arg < 0.0) {
            arg = ceil(arg - 0.5);
        } else {
            arg = 0.0;
        }
        sv_setnv(ST(0), (double)arg);    /* XXXXXX */
    }
    XSRETURN(1);
}
```

Notice the two lines commented with "XXXXXX". If you check the first section of the typemap file, you'll see that doubles are of type T_DOUBLE. In the INPUT section, an argument that is T_DOUBLE is assigned to the variable arg by calling the routine SvNV on something, then casting it to double, then assigned to the variable arg. Similarly, in the OUTPUT section, once arg has its final value, it is passed to the sv_setnv function to be passed back to the calling subroutine. These two functions are explained in [perlguts](#); we'll talk more later about what that "ST(0)" means in the section on the argument stack.

Warning about Output Arguments

In general, it's not a good idea to write extensions that modify their input parameters, as in Example 3. Instead, you should probably return multiple values in an array and let the caller handle them (we'll do this in a later example). However, in order to better accommodate calling pre-existing C routines, which often do modify their input parameters, this behavior is tolerated.

EXAMPLE 4

In this example, we'll now begin to write XSUBs that will interact with pre-defined C libraries. To begin with, we will build a small library of our own, then let h2xs write our .pm and .xs files for us.

Create a new directory called Mytest2 at the same level as the directory Mytest. In the Mytest2 directory, create another directory called mylib, and cd into that directory.

Here we'll create some files that will generate a test library. These will include a C source file and a header file. We'll also create a Makefile.PL in this directory. Then we'll make sure that running make at the Mytest2 level will automatically run this Makefile.PL file and the resulting Makefile.

In the mylib directory, create a file mylib.h that looks like this:

```
#define TESTVAL 4

extern double    foo(int, long, const char*);
```

Also create a file mylib.c that looks like this:

```
#include <stdlib.h>
#include "../mylib.h"

double
foo(int a, long b, const char *c)
{
    return (a + b + atof(c) + TESTVAL);
}
```



```
}
```

And finally create a file Makefile.PL that looks like this:

```
use ExtUtils::MakeMaker;
$Verbose = 1;
WriteMakefile(
    NAME      => 'Mytest2::mylib',
    SKIP      => [qw(all static static_lib dynamic dynamic_lib)],
    clean     => {'FILES' => 'libmylib$(LIB_EXT)'},
);

sub MY::top_targets {
    ,

all :: static

pure_all :: static

static ::      libmylib$(LIB_EXT)

libmylib$(LIB_EXT): $(O_FILES)
    $(AR) cr libmylib$(LIB_EXT) $(O_FILES)
    $(RANLIB) libmylib$(LIB_EXT)

';
}
```

Make sure you use a tab and not spaces on the lines beginning with "\$ (AR) " and "\$ (RANLIB) ". Make will not function properly if you use spaces. It has also been reported that the "cr" argument to \$ (AR) is unnecessary on Win32 systems.

We will now create the main top-level Mytest2 files. Change to the directory above Mytest2 and run the following command:

```
% h2xs -O -n Mytest2 ./Mytest2/mylib/mylib.h
```

This will print out a warning about overwriting Mytest2, but that's okay. Our files are stored in Mytest2/mylib, and will be untouched.

The normal Makefile.PL that h2xs generates doesn't know about the mylib directory. We need to tell it that there is a subdirectory and that we will be generating a library in it. Let's add the argument MYEXTLIB to the WriteMakefile call so that it looks like this:

```
WriteMakefile(
    'NAME'      => 'Mytest2',
    'VERSION_FROM' => 'Mytest2.pm', # finds $VERSION
    'LIBS'      => [''],           # e.g., '-lm'
    'DEFINE'    => '',             # e.g., '-DHAVE_SOMETHING'
    'INC'       => '',             # e.g., '-I/usr/include/other'
    'MYEXTLIB'  => 'mylib/libmylib$(LIB_EXT)',
);
```

and then at the end add a subroutine (which will override the pre-existing subroutine). Remember to use a tab character to indent the line beginning with "cd"!

```
sub MY::postamble {
    ,

$(MYEXTLIB): mylib/Makefile
    cd mylib && $(MAKE) $(PASSTHRU)

';
}
```

Let's also fix the MANIFEST file so that it accurately reflects the contents of our extension. The single line that says "mylib" should be replaced by the following three lines:

```
mylib/Makefile.PL
mylib/mylib.c
mylib/mylib.h
```

To keep our namespace nice and unpolluted, edit the .pm file and change the variable @EXPORT to @EXPORT_OK. Finally, in the .xs file, edit the #include line to read:

```
#include "mylib/mylib.h"
```

And also add the following function definition to the end of the .xs file:

```
double
foo(a,b,c)
    int      a
    long     b
    const char * c
    OUTPUT:
    RETVAL
```

Now we also need to create a typemap file because the default Perl doesn't currently support the const char * type. Create a file called typemap in the Mytest2 directory and place the following in it:

```
const char *      T_PV
```

Now run perl on the top-level Makefile.PL. Notice that it also created a Makefile in the mylib directory. Run make and watch that it does cd into the mylib directory and run make in there as well.

Now edit the test.pl script and change the BEGIN block to print "1.4", and add the following lines to the end of the script:

```
print &Mytest2::foo(1, 2, "Hello, world!") == 7 ? "ok 2\n" : "not ok 2\n";
print &Mytest2::foo(1, 2, "0.0") == 7 ? "ok 3\n" : "not ok 3\n";
print abs(&Mytest2::foo(0, 0, "-3.4") - 0.6) <= 0.01 ? "ok 4\n" : "not ok 4\n";
```

(When dealing with floating-point comparisons, it is best to not check for equality, but rather that the difference between the expected and actual result is below a certain amount (called epsilon) which is 0.01 in this case)

Run "make test" and all should be well.

What has happened here?

Unlike previous examples, we've now run h2xs on a real include file. This has caused some extra goodies to appear in both the .pm and .xs files.

- In the .xs file, there's now a #include directive with the absolute path to the mylib.h header file. We changed this to a relative path so that we could move the extension directory if we wanted to.
- There's now some new C code that's been added to the .xs file. The purpose of the constant routine is to make the values that are #define'd in the header file accessible by the Perl script (by calling either TESTVAL or &Mytest2::TESTVAL). There's also some XS code to allow calls to the constant routine.
- The .pm file originally exported the name TESTVAL in the @EXPORT array. This could lead to name clashes. A good rule of thumb is that if the #define is only going to be used by the C routines themselves, and not by the user, they should be removed from the @EXPORT array. Alternately, if you don't mind using the "fully qualified name" of a variable, you could move most or all of the items from the @EXPORT array into the @EXPORT_OK array.

- If our include file had contained `#include` directives, these would not have been processed by `h2xs`. There is no good solution to this right now.
- We've also told Perl about the library that we built in the `mylib` subdirectory. That required only the addition of the `MYEXTLIB` variable to the `WriteMakefile` call and the replacement of the postamble subroutine to `cd` into the subdirectory and run `make`. The `Makefile.PL` for the library is a bit more complicated, but not excessively so. Again we replaced the postamble subroutine to insert our own code. This code simply specified that the library to be created here was a static archive library (as opposed to a dynamically loadable library) and provided the commands to build it.

Anatomy of .xs file

The `.xs` file of *"EXAMPLE 4"* contained some new elements. To understand the meaning of these elements, pay attention to the line which reads

```
MODULE = Mytest2                PACKAGE = Mytest2
```

Anything before this line is plain C code which describes which headers to include, and defines some convenience functions. No translations are performed on this part, apart from having embedded POD documentation skipped over (see *perlpod*) it goes into the generated output C file as is.

Anything after this line is the description of XSUB functions. These descriptions are translated by `xsubpp` into C code which implements these functions using Perl calling conventions, and which makes these functions visible from Perl interpreter.

Pay a special attention to the function `constant`. This name appears twice in the generated `.xs` file: once in the first part, as a static C function, the another time in the second part, when an XSUB interface to this static C function is defined.

This is quite typical for `.xs` files: usually the `.xs` file provides an interface to an existing C function. Then this C function is defined somewhere (either in an external library, or in the first part of `.xs` file), and a Perl interface to this function (i.e. "Perl glue") is described in the second part of `.xs` file. The situation in *"EXAMPLE 1"*, *"EXAMPLE 2"*, and *"EXAMPLE 3"*, when all the work is done inside the "Perl glue", is somewhat of an exception rather than the rule.

Getting the fat out of XSUBs

In *"EXAMPLE 4"* the second part of `.xs` file contained the following description of an XSUB:

```
double
foo(a,b,c)
    int      a
    long     b
    const char * c
    OUTPUT:
    RETVAL
```

Note that in contrast with *"EXAMPLE 1"*, *"EXAMPLE 2"* and *"EXAMPLE 3"*, this description does not contain the actual *code* for what is done during a call to Perl function `foo()`. To understand what is going on here, one can add a `CODE` section to this XSUB:

```
double
foo(a,b,c)
    int      a
    long     b
    const char * c
    CODE:
    RETVAL = foo(a,b,c);
    OUTPUT:
    RETVAL
```

However, these two XSUBs provide almost identical generated C code: **xsubpp** compiler is smart enough to figure out the `CODE:` section from the first two lines of the description of XSUB. What about `OUTPUT:` section? In fact, that is absolutely the same! The `OUTPUT:` section can be removed as well, *as far as `CODE:` section or `PPCODE:` section is not specified*: **xsubpp** can see that it needs to generate a function call section, and will autogenerate the `OUTPUT` section too. Thus one can shortcut the XSUB to become:

```
double
foo(a,b,c)
    int      a
    long     b
    const char * c
```

Can we do the same with an XSUB

```
int
is_even(input)
    int      input
CODE:
    RETVAL = (input % 2 == 0);
OUTPUT:
    RETVAL
```

of *"EXAMPLE 2"*? To do this, one needs to define a C function `int is_even(int input)`. As we saw in *Anatomy of .xs file*, a proper place for this definition is in the first part of .xs file. In fact a C function

```
int
is_even(int arg)
{
    return (arg % 2 == 0);
}
```

is probably overkill for this. Something as simple as a `#define` will do too:

```
#define is_even(arg)    ((arg) % 2 == 0)
```

After having this in the first part of .xs file, the "Perl glue" part becomes as simple as

```
int
is_even(input)
    int      input
```

This technique of separation of the glue part from the workhorse part has obvious tradeoffs: if you want to change a Perl interface, you need to change two places in your code. However, it removes a lot of clutter, and makes the workhorse part independent from idiosyncrasies of Perl calling convention. (In fact, there is nothing Perl-specific in the above description, a different version of **xsubpp** might have translated this to TCL glue or Python glue as well.)

More about XSUB arguments

With the completion of Example 4, we now have an easy way to simulate some real-life libraries whose interfaces may not be the cleanest in the world. We shall now continue with a discussion of the arguments passed to the **xsubpp** compiler.

When you specify arguments to routines in the .xs file, you are really passing three pieces of information for each argument listed. The first piece is the order of that argument relative to the others (first, second, etc). The second is the type of argument, and consists of the type declaration of the argument (e.g., `int`, `char*`, etc). The third piece is the calling convention for the argument in the call to the library function.

While Perl passes arguments to functions by reference, C passes arguments by value; to implement a C function which modifies data of one of the "arguments", the actual argument of this C function would be a pointer to the data. Thus two C functions with declarations

```
int string_length(char *s);
int upper_case_char(char *cp);
```

may have completely different semantics: the first one may inspect an array of chars pointed by *s*, and the second one may immediately dereference *cp* and manipulate **cp* only (using the return value as, say, a success indicator). From Perl one would use these functions in a completely different manner.

One conveys this info to **xsubpp** by replacing *** before the argument by *&*. *&* means that the argument should be passed to a library function by its address. The above two function may be XSUB-ified as

```
int
string_length(s)
    char *  s

int
upper_case_char(cp)
    char    &cp
```

For example, consider:

```
int
foo(a,b)
    char    &a
    char *  b
```

The first Perl argument to this function would be treated as a char and assigned to the variable *a*, and its address would be passed into the function *foo*. The second Perl argument would be treated as a string pointer and assigned to the variable *b*. The *value* of *b* would be passed into the function *foo*. The actual call to the function *foo* that **xsubpp** generates would look like this:

```
foo(&a, b);
```

xsubpp will parse the following function argument lists identically:

```
char    &a
char&a
char    & a
```

However, to help ease understanding, it is suggested that you place a "&" next to the variable name and away from the variable type), and place a "*" near the variable type, but away from the variable name (as in the call to *foo* above). By doing so, it is easy to understand exactly what will be passed to the C function — it will be whatever is in the "last column".

You should take great pains to try to pass the function the type of variable it wants, when possible. It will save you a lot of trouble in the long run.

The Argument Stack

If we look at any of the C code generated by any of the examples except example 1, you will notice a number of references to *ST(n)*, where *n* is usually 0. "ST" is actually a macro that points to the *n*'th argument on the argument stack. *ST(0)* is thus the first argument on the stack and therefore the first argument passed to the XSUB, *ST(1)* is the second argument, and so on.

When you list the arguments to the XSUB in the .xs file, that tells **xsubpp** which argument corresponds to which of the argument stack (i.e., the first one listed is the first argument, and so on). You invite disaster if you do not list them in the same order as the function expects them.

The actual values on the argument stack are pointers to the values passed in. When an argument is listed as being an OUTPUT value, its corresponding value on the stack (i.e., *ST(0)* if it was the first argument) is changed. You can verify this by looking at the C code generated for Example 3. The code for the *round()* XSUB routine contains lines that look like this:

```
double arg = (double) SvNV(ST(0));
```

```
/* Round the contents of the variable arg */
sv_setnv(ST(0), (double)arg);
```

The `arg` variable is initially set by taking the value from `ST(0)`, then is stored back into `ST(0)` at the end of the routine.

XSUBs are also allowed to return lists, not just scalars. This must be done by manipulating stack values `ST(0)`, `ST(1)`, etc, in a subtly different way. See [perlx](#) for details.

XSUBs are also allowed to avoid automatic conversion of Perl function arguments to C function arguments. See [perlx](#) for details. Some people prefer manual conversion by inspecting `ST(i)` even in the cases when automatic conversion will do, arguing that this makes the logic of an XSUB call clearer. Compare with *"Getting the fat out of XSUBs"* for a similar tradeoff of a complete separation of "Perl glue" and "workhorse" parts of an XSUB.

While experts may argue about these idioms, a novice to Perl guts may prefer a way which is as little Perl-guts-specific as possible, meaning automatic conversion and automatic call generation, as in *"Getting the fat out of XSUBs"*. This approach has the additional benefit of protecting the XSUB writer from future changes to the Perl API.

Extending your Extension

Sometimes you might want to provide some extra methods or subroutines to assist in making the interface between Perl and your extension simpler or easier to understand. These routines should live in the `.pm` file. Whether they are automatically loaded when the extension itself is loaded or only loaded when called depends on where in the `.pm` file the subroutine definition is placed. You can also consult [AutoLoader](#) for an alternate way to store and load your extra subroutines.

Documenting your Extension

There is absolutely no excuse for not documenting your extension. Documentation belongs in the `.pm` file. This file will be fed to `pod2man`, and the embedded documentation will be converted to the manpage format, then placed in the `blib` directory. It will be copied to Perl's manpage directory when the extension is installed.

You may intersperse documentation and Perl code within the `.pm` file. In fact, if you want to use method autoloading, you must do this, as the comment inside the `.pm` file explains.

See [perlpod](#) for more information about the pod format.

Installing your Extension

Once your extension is complete and passes all its tests, installing it is quite simple: you simply run "make install". You will either need to have write permission into the directories where Perl is installed, or ask your system administrator to run the make for you.

Alternately, you can specify the exact directory to place the extension's files by placing a `"PREFIX=/destination/directory"` after the `make install`. (or in between the `make` and `install` if you have a brain-dead version of `make`). This can be very useful if you are building an extension that will eventually be distributed to multiple systems. You can then just archive the files in the destination directory and distribute them to your destination systems.

EXAMPLE 5

In this example, we'll do some more work with the argument stack. The previous examples have all returned only a single value. We'll now create an extension that returns an array.

This extension is very Unix-oriented (struct `statfs` and the `statfs` system call). If you are not running on a Unix system, you can substitute for `statfs` any other function that returns multiple values, you can hard-code values to be returned to the caller (although this will be a bit harder to test the error case), or you can simply not do this example. If you change the XSUB, be sure to fix the test cases to match the changes.

Return to the `Mytest` directory and add the following code to the end of `Mytest.xs`:

```

void
statfs(path)
    char * path
INIT:
    int i;
    struct statfs buf;

PPCODE:
    i = statfs(path, &buf);
    if (i == 0) {
        XPUSHs(sv_2mortal(newSVnv(buf.f_bavail)));
        XPUSHs(sv_2mortal(newSVnv(buf.f_bfree)));
        XPUSHs(sv_2mortal(newSVnv(buf.f_blocks)));
        XPUSHs(sv_2mortal(newSVnv(buf.f_bsize)));
        XPUSHs(sv_2mortal(newSVnv(buf.f_ffree)));
        XPUSHs(sv_2mortal(newSVnv(buf.f_files)));
        XPUSHs(sv_2mortal(newSVnv(buf.f_type)));
        XPUSHs(sv_2mortal(newSVnv(buf.f_fsid[0])));
        XPUSHs(sv_2mortal(newSVnv(buf.f_fsid[1])));
    } else {
        XPUSHs(sv_2mortal(newSVnv(errno)));
    }

```

You'll also need to add the following code to the top of the .xs file, just after the include of "XSUB.h":

```
#include <sys/vfs.h>
```

Also add the following code segment to test.pl while incrementing the "1..9" string in the BEGIN block to "1..11":

```

@a = &Mytest::statfs("/blech");
print ((scalar(@a) == 1 && $a[0] == 2) ? "ok 10\n" : "not ok 10\n");
@a = &Mytest::statfs("/");
print scalar(@a) == 9 ? "ok 11\n" : "not ok 11\n";

```

New Things in this Example

This example added quite a few new concepts. We'll take them one at a time.

- The INIT: directive contains code that will be placed immediately after the argument stack is decoded. C does not allow variable declarations at arbitrary locations inside a function, so this is usually the best way to declare local variables needed by the XSUB. (Alternatively, one could put the whole PPCODE: section into braces, and put these declarations on top.)
- This routine also returns a different number of arguments depending on the success or failure of the call to statfs. If there is an error, the error number is returned as a single-element array. If the call is successful, then a 9-element array is returned. Since only one argument is passed into this function, we need room on the stack to hold the 9 values which may be returned.

We do this by using the PPCODE: directive, rather than the CODE: directive. This tells **xsubpp** that we will be managing the return values that will be put on the argument stack by ourselves.

- When we want to place values to be returned to the caller onto the stack, we use the series of macros that begin with "XPUSH". There are five different versions, for placing integers, unsigned integers, doubles, strings, and Perl scalars on the stack. In our example, we placed a Perl scalar onto the stack. (In fact this is the only macro which can be used to return multiple values.)

The XPUSH* macros will automatically extend the return stack to prevent it from being overrun. You push values onto the stack in the order you want them seen by the calling program.

- The values pushed onto the return stack of the XSUB are actually mortal SV's. They are made mortal so that once the values are copied by the calling program, the SV's that held the returned values can be deallocated. If they were not mortal, then they would continue to exist after the XSUB routine returned, but would not be accessible. This is a memory leak.
- If we were interested in performance, not in code compactness, in the success branch we would not use XPUSHs macros, but PUSHs macros, and would pre-extend the stack before pushing the return values:

```
EXTEND(SP, 9);
```

The tradeoff is that one needs to calculate the number of return values in advance (though overextending the stack will not typically hurt anything but memory consumption).

Similarly, in the failure branch we could use PUSHs *without* extending the stack: the Perl function reference comes to an XSUB on the stack, thus the stack is *always* large enough to take one return value.

EXAMPLE 6

In this example, we will accept a reference to an array as an input parameter, and return a reference to an array of hashes. This will demonstrate manipulation of complex Perl data types from an XSUB.

This extension is somewhat contrived. It is based on the code in the previous example. It calls the statfs function multiple times, accepting a reference to an array of filenames as input, and returning a reference to an array of hashes containing the data for each of the filesystems.

Return to the Mytest directory and add the following code to the end of Mytest.xs:

```
SV *
multi_statfs(paths)
    SV * paths
INIT:
    AV * results;
    I32 numpaths = 0;
    int i, n;
    struct statfs buf;

    if ((!SvROK(paths))
        || (SvTYPE(SvRV(paths)) != SVt_PVAV)
        || ((numpaths = av_len((AV *)SvRV(paths))) < 0))
    {
        XSRETURN_UNDEF;
    }
    results = (AV *)sv_2mortal((SV *)newAV());
CODE:
    for (n = 0; n <= numpaths; n++) {
        HV * rh;
        STRLEN l;
        char * fn = SvPV(*av_fetch((AV *)SvRV(paths), n, 0), 1);

        i = statfs(fn, &buf);
        if (i != 0) {
            av_push(results, newSVnv(errno));
            continue;
        }

        rh = (HV *)sv_2mortal((SV *)newHV());
        hv_store(rh, "f_bavail", 8, newSVnv(buf.f_bavail), 0);
        hv_store(rh, "f_bfree", 7, newSVnv(buf.f_bfree), 0);
```



```

    hv_store(rh, "f_blocks", 8, newSVnv(buf.f_blocks), 0);
    hv_store(rh, "f_bsize", 7, newSVnv(buf.f_bsize), 0);
    hv_store(rh, "f_ffree", 7, newSVnv(buf.f_ffree), 0);
    hv_store(rh, "f_files", 7, newSVnv(buf.f_files), 0);
    hv_store(rh, "f_type", 6, newSVnv(buf.f_type), 0);

    av_push(results, newRV((SV *)rh));
}
RETVAL = newRV((SV *)results);
OUTPUT:
RETVAL

```

And add the following code to test.pl, while incrementing the "1..11" string in the BEGIN block to "1..13":

```

$results = Mytest::multi_statfs([ '/', '/blech' ]);
print ((ref $results->[0]) ? "ok 12\n" : "not ok 12\n");
print ((! ref $results->[1]) ? "ok 13\n" : "not ok 13\n");

```

New Things in this Example

There are a number of new concepts introduced here, described below:

- This function does not use a typemap. Instead, we declare it as accepting one SV* (scalar) parameter, and returning an SV* value, and we take care of populating these scalars within the code. Because we are only returning one value, we don't need a PPCODE: directive – instead, we use CODE: and OUTPUT: directives.
- When dealing with references, it is important to handle them with caution. The INIT: block first checks that SvROK returns true, which indicates that paths is a valid reference. It then verifies that the object referenced by paths is an array, using SvRV to dereference paths, and SvTYPE to discover its type. As an added test, it checks that the array referenced by paths is non-empty, using the av_len function (which returns -1 if the array is empty). The XSRETURN_UNDEF macro is used to abort the XSUB and return the undefined value whenever all three of these conditions are not met.
- We manipulate several arrays in this XSUB. Note that an array is represented internally by an AV* pointer. The functions and macros for manipulating arrays are similar to the functions in Perl: av_len returns the highest index in an AV*, much like \$#array; av_fetch fetches a single scalar value from an array, given its index; av_push pushes a scalar value onto the end of the array, automatically extending the array as necessary.

Specifically, we read pathnames one at a time from the input array, and store the results in an output array (results) in the same order. If statfs fails, the element pushed onto the return array is the value of errno after the failure. If statfs succeeds, though, the value pushed onto the return array is a reference to a hash containing some of the information in the statfs structure.

As with the return stack, it would be possible (and a small performance win) to pre-extend the return array before pushing data into it, since we know how many elements we will return:

```

av_extend(results, numpaths);

```

- We are performing only one hash operation in this function, which is storing a new scalar under a key using hv_store. A hash is represented by an HV* pointer. Like arrays, the functions for manipulating hashes from an XSUB mirror the functionality available from Perl. See [perlguts](#) and [perlapi](#) for details.
- To create a reference, we use the newRV function. Note that you can cast an AV* or an HV* to type SV* in this case (and many others). This allows you to take references to arrays, hashes and scalars with the same function. Conversely, the SvRV function always returns an SV*, which may need to be cast to the appropriate type if it is something other than a scalar (check with SvTYPE).

- At this point, xsubpp is doing very little work – the differences between Mytest.xs and Mytest.c are minimal.

EXAMPLE 7 (Coming Soon)

XPUSH args AND set RETVAL AND assign return value to array

EXAMPLE 8 (Coming Soon)

Setting \$!

EXAMPLE 9 Passing open files to XSes

You would think passing files to an XS is difficult, with all the typeglobs and stuff. Well, it isn't.

Suppose that for some strange reason we need a wrapper around the standard C library function `fputs()`. This is all we need:

```
#define PERLIO_NOT_STDIO 0
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include <stdio.h>

int
fputs(s, stream)
    char *      s
    FILE *      stream
```

The real work is done in the standard typemap.

But you lose all the fine stuff done by the perlio layers. This calls the stdio function `fputs()`, which knows nothing about them.

The standard typemap offers three variants of `PerlIO *`: `InputStream (T_IN)`, `InOutputStream (T_INOUT)` and `OutputStream (T_OUT)`. A bare `PerlIO *` is considered a `T_INOUT`. If it matters in your code (see below for why it might) `#define` or `typedef` one of the specific names and use that as the argument or result type in your XS file.

The standard typemap does not contain `PerlIO *` before perl 5.7, but it has the three stream variants. Using a `PerlIO *` directly is not backwards compatible unless you provide your own typemap.

For streams coming *from* perl the main difference is that `OutputStream` will get the output `PerlIO *` – which may make a difference on a socket. Like in our example...

For streams being handed *to* perl a new file handle is created (i.e. a reference to a new glob) and associated with the `PerlIO *` provided. If the read/write state of the `PerlIO *` is not correct then you may get errors or warnings from when the file handle is used. So if you opened the `PerlIO *` as "w" it should really be an `OutputStream` if open as "r" it should be an `InputStream`.

Now, suppose you want to use perlio layers in your XS. We'll use the perlio `PerlIO_puts()` function as an example.

In the C part of the XS file (above the first `MODULE` line) you have

```
#define OutputStream    PerlIO *
or
typedef PerlIO *        OutputStream;
```

And this is the XS code:

```
int
perlioputs(s, stream)
    char *      s
```

```

        OutputStream      stream
CODE:
        RETVAL = PerlIO_puts(stream, s);
OUTPUT:
        RETVAL

```

We have to use a CODE section because `PerlIO_puts()` has the arguments reversed compared to `fputs()`, and we want to keep the arguments the same.

Wanting to explore this thoroughly, we want to use the stdio `fputs()` on a `PerlIO *`. This means we have to ask the perlio system for a stdio `FILE *`:

```

        int
        perliofputs(s, stream)
            char *          s
            OutputStream      stream
PREINIT:
        FILE *fp = PerlIO_findFILE(stream);
CODE:
        if (fp != (FILE*) 0) {
            RETVAL = fputs(s, fp);
        } else {
            RETVAL = -1;
        }
OUTPUT:
        RETVAL

```

Note: `PerlIO_findFILE()` will search the layers for a stdio layer. If it can't find one, it will call `PerlIO_exportFILE()` to generate a new stdio `FILE`. Please only call `PerlIO_exportFILE()` if you want a *new* `FILE`. It will generate one on each call and push a new stdio layer. So don't call it repeatedly on the same file. `PerlIO_findFILE` will retrieve the stdio layer once it has been generated by `PerlIO_exportFILE()`.

This applies to the perlio system only. For versions before 5.7, `PerlIO_exportFILE()` is equivalent to `PerlIO_findFILE()`.

Troubleshooting these Examples

As mentioned at the top of this document, if you are having problems with these example extensions, you might see if any of these help you.

- In versions of 5.002 prior to the gamma version, the test script in Example 1 will not function properly. You need to change the "use lib" line to read:

```
use lib './blib';
```

- In versions of 5.002 prior to version 5.002b1h, the test.pl file was not automatically created by h2xs. This means that you cannot say "make test" to run the test script. You will need to add the following line before the "use extension" statement:

```
use lib './blib';
```

- In versions 5.000 and 5.001, instead of using the above line, you will need to use the following line:

```
BEGIN { unshift(@INC, "./blib") }
```

- This document assumes that the executable named "perl" is Perl version 5. Some systems may have installed Perl version 5 as "perl5".

See also

For more information, consult [perlguts](#), [perlapi](#), [perlxs](#), [perlmod](#), and [perlpod](#).

Author

Jeff Okamoto <okamoto@corp.hp.com>

Reviewed and assisted by Dean Roehrich, Ilya Zakharevich, Andreas Koenig, and Tim Bunce.

PerlIO material contributed by Lupe Christoph, with some clarification by Nick Ing-Simmons.

Last Changed

2002/05/08

NAME

AnyDBM_File – provide framework for multiple DBMs

NDBM_File, DB_File, GDBM_File, SDBM_File, ODBM_File – various DBM implementations

SYNOPSIS

```
use AnyDBM_File;
```

DESCRIPTION

This module is a "pure virtual base class"—it has nothing of its own. It's just there to inherit from one of the various DBM packages. It prefers ndbm for compatibility reasons with Perl 4, then Berkeley DB (See [DB_File](#)), GDBM, SDBM (which is always there—it comes with Perl), and finally ODBM. This way old programs that used to use NDBM via dbmopen() can still do so, but new ones can reorder @ISA:

```
BEGIN { @AnyDBM_File::ISA = qw(DB_File GDBM_File NDBM_File) }
use AnyDBM_File;
```

Having multiple DBM implementations makes it trivial to copy database formats:

```
use POSIX; use NDBM_File; use DB_File;
tie %newhash, 'DB_File', $new_filename, O_CREAT|O_RDWR;
tie %oldhash, 'NDBM_File', $old_filename, 1, 0;
%newhash = %oldhash;
```

DBM Comparisons

Here's a partial table of features the different packages offer:

	odbm	ndbm	sdbm	gdbm	bsd-db
	----	----	----	----	-----
Linkage comes w/ perl	yes	yes	yes	yes	yes
Src comes w/ perl	no	no	yes	no	no
Comes w/ many unix os	yes	yes[0]	no	no	no
Builds ok on !unix	?	?	yes	yes	?
Code Size	?	?	small	big	big
Database Size	?	?	small	big?	ok[1]
Speed	?	?	slow	ok	fast
FTPable	no	no	yes	yes	yes
Easy to build	N/A	N/A	yes	yes	ok[2]
Size limits	1k	4k	1k[3]	none	none
Byte-order independent	no	no	no	no	yes
Licensing restrictions	?	?	no	yes	no

[0] on mixed universe machines, may be in the bsd compat library, which is often shunned.

[1] Can be trimmed if you compile for one access method.

[2] See [DB_File](#). Requires symbolic links.

[3] By default, but can be redefined.

SEE ALSO

dbm(3), ndbm(3), DB_File(3), [perldbmfilter](#)

NAME

Attribute::Handlers – Simpler definition of attribute handlers

VERSION

This document describes version 0.77 of Attribute::Handlers, released June 8, 2002.

SYNOPSIS

```
package MyClass;
require v5.6.0;
use Attribute::Handlers;
no warnings 'redefine';

sub Good : ATTR(SCALAR) {
    my ($package, $symbol, $referent, $attr, $data) = @_;

    # Invoked for any scalar variable with a :Good attribute,
    # provided the variable was declared in MyClass (or
    # a derived class) or typed to MyClass.

    # Do whatever to $referent here (executed in CHECK phase).
    ...
}

sub Bad : ATTR(SCALAR) {
    # Invoked for any scalar variable with a :Bad attribute,
    # provided the variable was declared in MyClass (or
    # a derived class) or typed to MyClass.
    ...
}

sub Good : ATTR(ARRAY) {
    # Invoked for any array variable with a :Good attribute,
    # provided the variable was declared in MyClass (or
    # a derived class) or typed to MyClass.
    ...
}

sub Good : ATTR(HASH) {
    # Invoked for any hash variable with a :Good attribute,
    # provided the variable was declared in MyClass (or
    # a derived class) or typed to MyClass.
    ...
}

sub Ugly : ATTR(CODE) {
    # Invoked for any subroutine declared in MyClass (or a
    # derived class) with an :Ugly attribute.
    ...
}

sub Omni : ATTR {
    # Invoked for any scalar, array, hash, or subroutine
    # with an :Omni attribute, provided the variable or
    # subroutine was declared in MyClass (or a derived class)
    # or the variable was typed to MyClass.
    # Use ref($_[2]) to determine what kind of referent it was.
    ...
}
```

```
use Attribute::Handlers autotie => { Cycle => Tie::Cycle };

my $next : Cycle(['A'..'Z']);
```

DESCRIPTION

This module, when inherited by a package, allows that package's class to define attribute handler subroutines for specific attributes. Variables and subroutines subsequently defined in that package, or in packages derived from that package may be given attributes with the same names as the attribute handler subroutines, which will then be called in one of the compilation phases (i.e. in a BEGIN, CHECK, INIT, or END block).

To create a handler, define it as a subroutine with the same name as the desired attribute, and declare the subroutine itself with the `attribute :ATTR`. For example:

```
package LoudDecl;
use Attribute::Handlers;

sub Loud :ATTR {
    my ($package, $symbol, $referent, $attr, $data, $phase) = @_;
    print STDERR
        ref($referent), " ",
        *{$symbol}{NAME}, " ",
        "($referent) ", "was just declared ",
        "and ascribed the ${attr} attribute ",
        "with data ($data)\n",
        "in phase $phase\n";
}
```

This creates a handler for the attribute `:Loud` in the class `LoudDecl`. Thereafter, any subroutine declared with a `:Loud` attribute in the class `LoudDecl`:

```
package LoudDecl;

sub foo: Loud {...}
```

causes the above handler to be invoked, and passed:

- [0] the name of the package into which it was declared;
- [1] a reference to the symbol table entry (typeglob) containing the subroutine;
- [2] a reference to the subroutine;
- [3] the name of the attribute;
- [4] any data associated with that attribute;
- [5] the name of the phase in which the handler is being invoked.

Likewise, declaring any variables with the `:Loud` attribute within the package:

```
package LoudDecl;

my $foo :Loud;
my @foo :Loud;
my %foo :Loud;
```

will cause the handler to be called with a similar argument list (except, of course, that `$_[2]` will be a reference to the variable).

The package name argument will typically be the name of the class into which the subroutine was declared, but it may also be the name of a derived class (since handlers are inherited).

If a lexical variable is given an attribute, there is no symbol table to which it belongs, so the symbol table argument (`$_[1]`) is set to the string `'LEXICAL'` in that case. Likewise, ascribing an attribute to an anonymous subroutine results in a symbol table argument of `'ANON'`.

The data argument passes in the value (if any) associated with the attribute. For example, if `&foo` had been declared:

```
sub foo :Loud("turn it up to 11, man!") {...}
```

then the string "turn it up to 11, man!" would be passed as the last argument.

Attribute::Handlers makes strenuous efforts to convert the data argument (`$_[4]`) to a useable form before passing it to the handler (but see *"Non-interpretive attribute handlers"*). For example, all of these:

```
sub foo :Loud(till=>ears=>are=>bleeding) {...}
sub foo :Loud(['till','ears','are','bleeding']) {...}
sub foo :Loud(qw/till ears are bleeding/) {...}
sub foo :Loud(qw/my, ears, are, bleeding/) {...}
sub foo :Loud(till,ears,are,bleeding) {...}
```

causes it to pass `['till','ears','are','bleeding']` as the handler's data argument. However, if the data can't be parsed as valid Perl, then it is passed as an uninterpreted string. For example:

```
sub foo :Loud(my,ears,are,bleeding) {...}
sub foo :Loud(qw/my ears are bleeding) {...}
```

cause the strings `'my,ears,are,bleeding'` and `'qw/my ears are bleeding'` respectively to be passed as the data argument.

If the attribute has only a single associated scalar data value, that value is passed as a scalar. If multiple values are associated, they are passed as an array reference. If no value is associated with the attribute, `undef` is passed.

Typed lexicals

Regardless of the package in which it is declared, if a lexical variable is ascribed an attribute, the handler that is invoked is the one belonging to the package to which it is typed. For example, the following declarations:

```
package OtherClass;

my LoudDecl $loudobj : Loud;
my LoudDecl @loudobjs : Loud;
my LoudDecl %loudobjex : Loud;
```

causes the `LoudDecl::Loud` handler to be invoked (even if `OtherClass` also defines a handler for `:Loud` attributes).

Type-specific attribute handlers

If an attribute handler is declared and the `:ATTR` specifier is given the name of a built-in type (`SCALAR`, `ARRAY`, `HASH`, or `CODE`), the handler is only applied to declarations of that type. For example, the following definition:

```
package LoudDecl;

sub RealLoud :ATTR(SCALAR) { print "Yeeeeow!" }
```

creates an attribute handler that applies only to scalars:

```
package Painful;
use base LoudDecl;

my $metal : RealLoud;           # invokes &LoudDecl::RealLoud
my @metal : RealLoud;           # error: unknown attribute
my %metal : RealLoud;           # error: unknown attribute
sub metal : RealLoud {...}      # error: unknown attribute
```

You can, of course, declare separate handlers for these types as well (but you'll need to specify no warnings `'redefine'` to do it quietly):


```

package LoudDecl;
use Attribute::Handlers;
no warnings 'redefine';

sub RealLoud :ATTR(SCALAR) { print "Yeeeeow!" }
sub RealLoud :ATTR(ARRAY) { print "Urrrrrrrrrr!" }
sub RealLoud :ATTR(HASH) { print "Arrrrrgggghhhhhh!" }
sub RealLoud :ATTR(CODE) { croak "Real loud sub torpedoed" }

```

You can also explicitly indicate that a single handler is meant to be used for all types of referents like so:

```

package LoudDecl;
use Attribute::Handlers;

sub SeriousLoud :ATTR(ANY) { warn "Hearing loss imminent" }

```

(I.e. `ATTR(ANY)` is a synonym for `:ATTR`).

Non-interpretive attribute handlers

Occasionally the strenuous efforts `Attribute::Handlers` makes to convert the data argument (`$_[4]`) to a useable form before passing it to the handler get in the way.

You can turn off that eagerness-to-help by declaring an attribute handler with the keyword `RAWDATA`. For example:

```

sub Raw          : ATTR(RAWDATA) { ... }
sub Nekkid       : ATTR(SCALAR, RAWDATA) { ... }
sub Au::Naturale : ATTR(RAWDATA, ANY) { ... }

```

Then the handler makes absolutely no attempt to interpret the data it receives and simply passes it as a string:

```

my $power : Raw(1..100);           # handlers receives "1..100"

```

Phase-specific attribute handlers

By default, attribute handlers are called at the end of the compilation phase (in a `CHECK` block). This seems to be optimal in most cases because most things that can be defined are defined by that point but nothing has been executed.

However, it is possible to set up attribute handlers that are called at other points in the program's compilation or execution, by explicitly stating the phase (or phases) in which you wish the attribute handler to be called. For example:

```

sub Early      :ATTR(SCALAR, BEGIN) { ... }
sub Normal     :ATTR(SCALAR, CHECK) { ... }
sub Late       :ATTR(SCALAR, INIT)  { ... }
sub Final      :ATTR(SCALAR, END)   { ... }
sub Bookends   :ATTR(SCALAR, BEGIN, END) { ... }

```

As the last example indicates, a handler may be set up to be (re)called in two or more phases. The phase name is passed as the handler's final argument.

Note that attribute handlers that are scheduled for the `BEGIN` phase are handled as soon as the attribute is detected (i.e. before any subsequently defined `BEGIN` blocks are executed).

Attributes as tie interfaces

Attributes make an excellent and intuitive interface through which to tie variables. For example:

```

use Attribute::Handlers;
use Tie::Cycle;

sub UNIVERSAL::Cycle : ATTR(SCALAR) {
    my ($package, $symbol, $referent, $attr, $data, $phase) = @_;

```

```

        $data = [ $data ] unless ref $data eq 'ARRAY';
        tie $$referent, 'Tie::Cycle', $data;
    }
    # and thereafter...

    package main;

    my $next : Cycle('A'..'Z');      # $next is now a tied variable

    while (<>) {
        print $next;
    }

```

Note that, because the `Cycle` attribute receives its arguments in the `$data` variable, if the attribute is given a list of arguments, `$data` will consist of a single array reference; otherwise, it will consist of the single argument directly. Since `Tie::Cycle` requires its cycling values to be passed as an array reference, this means that we need to wrap non-array-reference arguments in an array constructor:

```
$data = [ $data ] unless ref $data eq 'ARRAY';
```

Typically, however, things are the other way around: the tieable class expects its arguments as a flattened list, so the attribute looks like:

```

sub UNIVERSAL::Cycle : ATTR(SCALAR) {
    my ($package, $symbol, $referent, $attr, $data, $phase) = @_;
    my @data = ref $data eq 'ARRAY' ? @$data : $data;
    tie $$referent, 'Tie::Whatever', @data;
}

```

This software pattern is so widely applicable that `Attribute::Handlers` provides a way to automate it: specifying `'autotie'` in the `use Attribute::Handlers` statement. So, the cycling example, could also be written:

```

use Attribute::Handlers autotie => { Cycle => 'Tie::Cycle' };
# and thereafter...

package main;

my $next : Cycle(['A'..'Z']);      # $next is now a tied variable

while (<>) {
    print $next;
}

```

Note that we now have to pass the cycling values as an array reference, since the `autotie` mechanism passes `tie` a list of arguments as a list (as in the `Tie::Whatever` example), *not* as an array reference (as in the original `Tie::Cycle` example at the start of this section).

The argument after `'autotie'` is a reference to a hash in which each key is the name of an attribute to be created, and each value is the class to which variables ascribed that attribute should be tied.

Note that there is no longer any need to import the `Tie::Cycle` module — `Attribute::Handlers` takes care of that automatically. You can even pass arguments to the module's `import` subroutine, by appending them to the class name. For example:

```

use Attribute::Handlers
    autotie => { Dir => 'Tie::Dir qw(DIR_UNLINK)' };

```

If the attribute name is unqualified, the attribute is installed in the current package. Otherwise it is installed in the qualifier's package:

```
package Here;
```

```

use Attribute::Handlers autotie => {
    Other::Good => Tie::SecureHash, # tie attr installed in Other::
    Bad => Tie::Taxes,             # tie attr installed in Here::
    UNIVERSAL::Ugly => Software::Patent # tie attr installed everywhere
};

```

Autoties are most commonly used in the module to which they actually tie, and need to export their attributes to any module that calls them. To facilitate this, Attribute::Handlers recognizes a special "pseudo-class" — `__CALLER__`, which may be specified as the qualifier of an attribute:

```

package Tie::Me::Kangaroo:Down::Sport;

use Attribute::Handlers autotie => { __CALLER__::Roo => __PACKAGE__ };

```

This causes Attribute::Handlers to define the `Roo` attribute in the package that imports the `Tie::Me::Kangaroo:Down::Sport` module.

=head3 Passing the tied object to tie

Occasionally it is important to pass a reference to the object being tied to the `TIESCALAR`, `TIEHASH`, etc. that ties it.

The `autotie` mechanism supports this too. The following code:

```

use Attribute::Handlers autotieref => { Selfish => Tie::Selfish };
my $var : Selfish(@args);

```

has the same effect as:

```

tie my $var, 'Tie::Selfish', @args;

```

But when "`autotieref`" is used instead of "`autotie`":

```

use Attribute::Handlers autotieref => { Selfish => Tie::Selfish };
my $var : Selfish(@args);

```

the effect is to pass the `tie` call an extra reference to the variable being tied:

```

tie my $var, 'Tie::Selfish', \$var, @args;

```

EXAMPLES

If the class shown in [SYNOPSIS](#) were placed in the `MyClass.pm` module, then the following code:

```

package main;
use MyClass;

my MyClass $slr :Good :Bad(1**1-1) :Omni(-vorous);

package SomeOtherClass;
use base MyClass;

sub tent { 'acle' }

sub fn :Ugly(sister) :Omni('po',tent()) {...}
my @arr :Good :Omni(s/cie/nt/);
my %hsh :Good(q/bye) :Omni(q/bus/);

```

would cause the following handlers to be invoked:

```

# my MyClass $slr :Good :Bad(1**1-1) :Omni(-vorous);
MyClass::Good:ATTR(SCALAR) ( 'MyClass',      # class
                             'LEXICAL',      # no typeglob
                             \$slr,          # referent
                             'Good',         # attr name
                             undef           # no attr data

```

```

        'CHECK',          # compiler phase
    );

MyClass::Bad:ATTR(SCALAR) ( 'MyClass',          # class
                             'LEXICAL',          # no typeglob
                             \$slr,              # referent
                             'Bad',              # attr name
                             0                   # eval'd attr data
                             'CHECK',            # compiler phase
    );

MyClass::Omni:ATTR(SCALAR) ( 'MyClass',          # class
                              'LEXICAL',          # no typeglob
                              \$slr,              # referent
                              'Omni',             # attr name
                              '-vorous'           # eval'd attr data
                              'CHECK',            # compiler phase
    );

# sub fn :Ugly(sister) :Omni('po',tent()) {...}

MyClass::UGLY:ATTR(CODE) ( 'SomeOtherClass',    # class
                           \*SomeOtherClass::fn, # typeglob
                           \&SomeOtherClass::fn, # referent
                           'Ugly',               # attr name
                           'sister'              # eval'd attr data
                           'CHECK',              # compiler phase
    );

MyClass::Omni:ATTR(CODE) ( 'SomeOtherClass',    # class
                           \*SomeOtherClass::fn, # typeglob
                           \&SomeOtherClass::fn, # referent
                           'Omni',               # attr name
                           ['po','acle']         # eval'd attr data
                           'CHECK',              # compiler phase
    );

# my @arr :Good :Omni(s/cie/nt/);

MyClass::Good:ATTR(ARRAY) ( 'SomeOtherClass',  # class
                             'LEXICAL',          # no typeglob
                             \@arr,              # referent
                             'Good',             # attr name
                             undef               # no attr data
                             'CHECK',            # compiler phase
    );

MyClass::Omni:ATTR(ARRAY) ( 'SomeOtherClass',  # class
                             'LEXICAL',          # no typeglob
                             \@arr,              # referent
                             'Omni',             # attr name
                             ""                  # eval'd attr data
                             'CHECK',            # compiler phase
    );

# my %hsh :Good(q/bye) :Omni(q/bus/);

MyClass::Good:ATTR(HASH) ( 'SomeOtherClass',   # class
                            'LEXICAL',           # no typeglob

```

```

                                \%hsh,          # referent
                                'Good',          # attr name
                                'q/bye'         # raw attr data
                                'CHECK',        # compiler phase
                                );

MyClass::Omni::ATTR(HASH) ( 'SomeOtherClass', # class
                             'LEXICAL',        # no typeglob
                             \%hsh,          # referent
                             'Omni',          # attr name
                             'bus'            # eval'd attr data
                             'CHECK',        # compiler phase
                             );

```

Installing handlers into UNIVERSAL, makes them...err..universal. For example:

```

package Descriptions;
use Attribute::Handlers;

my %name;
sub name { return $name{$_[2]} || *{$_[1]}{NAME} }

sub UNIVERSAL::Name :ATTR {
    $name{$_[2]} = $_[4];
}

sub UNIVERSAL::Purpose :ATTR {
    print STDERR "Purpose of ", &name, " is $_[4]\n";
}

sub UNIVERSAL::Unit :ATTR {
    print STDERR &name, " measured in $_[4]\n";
}

```

Let's you write:

```

use Descriptions;

my $capacity : Name(capacity)
              : Purpose(to store max storage capacity for files)
              : Unit(Gb);

package Other;

sub foo : Purpose(to foo all data before barring it) { }

# etc.

```

DIAGNOSTICS

Bad attribute type: ATTR(%s)

An attribute handler was specified with an `:ATTR(ref_type)`, but the type of referent it was defined to handle wasn't one of the five permitted: SCALAR, ARRAY, HASH, CODE, or ANY.

Attribute handler %s doesn't handle %s attributes

A handler for attributes of the specified name *was* defined, but not for the specified type of declaration. Typically encountered whe trying to apply a VAR attribute handler to a subroutine, or a SCALAR attribute handler to some other type of variable.

Declaration of %s attribute in package %s may clash with future reserved word

A handler for an attributes with an all-lowercase name was declared. An attribute with an

all-lowercase name might have a meaning to Perl itself some day, even though most don't yet. Use a mixed-case attribute name, instead.

Can't have two ATTR specifiers on one subroutine

You just can't, okay? Instead, put all the specifications together with commas between them in a single `ATTR(specification)`.

Can't autotie a %s

You can only declare autoties for types "SCALAR", "ARRAY", and "HASH". They're the only things (apart from typeglobs — which are not declarable) that Perl can tie.

Internal error: %s symbol went missing

Something is rotten in the state of the program. An attributed subroutine ceased to exist between the point it was declared and the point at which its attribute handler(s) would have been called.

Won't be able to apply END handler

You have defined an END handler for an attribute that is being applied to a lexical variable. Since the variable may not be available during END this won't happen.

AUTHOR

Damian Conway (damian@conway.org)

BUGS

There are undoubtedly serious bugs lurking somewhere in code this funky :-) Bug reports and other feedback are most welcome.

COPYRIGHT

Copyright (c) 2001, Damian Conway. All Rights Reserved.
This module is free software. It may be used, redistributed
and/or modified under the same terms as Perl itself.

NAME

attributes – get/set subroutine or variable attributes

SYNOPSIS

```
sub foo : method ;
my ($x,@y,%z) : Bent = 1;
my $s = sub : method { ... };

use attributes ();      # optional, to get subroutine declarations
my @attrlist = attributes::get(\&foo);

use attributes 'get'; # import the attributes::get subroutine
my @attrlist = get \&foo;
```

DESCRIPTION

Subroutine declarations and definitions may optionally have attribute lists associated with them. (Variable my declarations also may, but see the warning below.) Perl handles these declarations by passing some information about the call site and the thing being declared along with the attribute list to this module. In particular, the first example above is equivalent to the following:

```
use attributes __PACKAGE__, \&foo, 'method';
```

The second example in the synopsis does something equivalent to this:

```
use attributes ();
my ($x,@y,%z);
attributes::->import(__PACKAGE__, \$x, 'Bent');
attributes::->import(__PACKAGE__, \@y, 'Bent');
attributes::->import(__PACKAGE__, \%z, 'Bent');
($x,@y,%z) = 1;
```

Yes, that's a lot of expansion.

WARNING: attribute declarations for variables are still evolving. The semantics and interfaces of such declarations could change in future versions. They are present for purposes of experimentation with what the semantics ought to be. Do not rely on the current implementation of this feature.

There are only a few attributes currently handled by Perl itself (or directly by this module, depending on how you look at it.) However, package-specific attributes are allowed by an extension mechanism. (See *"Package-specific Attribute Handling"* below.)

The setting of subroutine attributes happens at compile time. Variable attributes in our declarations are also applied at compile time. However, my variables get their attributes applied at run-time. This means that you have to *reach* the run-time component of the my before those attributes will get applied. For example:

```
my $x : Bent = 42 if 0;
```

will neither assign 42 to \$x *nor* will it apply the Bent attribute to the variable.

An attempt to set an unrecognized attribute is a fatal error. (The error is trappable, but it still stops the compilation within that eval.) Setting an attribute with a name that's all lowercase letters that's not a built-in attribute (such as "foo") will result in a warning with `-w` or use warnings `'reserved'`.

Built-in Attributes

The following are the built-in attributes for subroutines:

locked

Setting this attribute is only meaningful when the subroutine or method is to be called by multiple threads. When set on a method subroutine (i.e., one marked with the **method** attribute below), Perl ensures that any invocation of it implicitly locks its first argument before execution. When set on a non-method subroutine, Perl ensures that a lock is taken on the subroutine itself before execution. The

semantics of the lock are exactly those of one explicitly taken with the `lock` operator immediately after the subroutine is entered.

method

Indicates that the referenced subroutine is a method. This has a meaning when taken together with the **locked** attribute, as described there. It also means that a subroutine so marked will not trigger the "Ambiguous call resolved as `CORE::%`" warning.

lvalue

Indicates that the referenced subroutine is a valid lvalue and can be assigned to. The subroutine must return a modifiable value such as a scalar variable, as described in [perlsub](#).

For global variables there is `unique` attribute: see [our](#).

Available Subroutines

The following subroutines are available for general use once this module has been loaded:

get This routine expects a single parameter—a reference to a subroutine or variable. It returns a list of attributes, which may be empty. If passed invalid arguments, it uses `die()` (via [Carp::croak/Carp](#)) to raise a fatal exception. If it can find an appropriate package name for a class method lookup, it will include the results from a `FETCH_type_ATTRIBUTES` call in its return list, as described in ["Package-specific Attribute Handling"](#) below. Otherwise, only [built-in attributes](#)/[Built-in Attributes](#) will be returned.

reftype

This routine expects a single parameter—a reference to a subroutine or variable. It returns the built-in type of the referenced variable, ignoring any package into which it might have been blessed. This can be useful for determining the *type* value which forms part of the method names described in ["Package-specific Attribute Handling"](#) below.

Note that these routines are *not* exported by default.

Package-specific Attribute Handling

WARNING: the mechanisms described here are still experimental. Do not rely on the current implementation. In particular, there is no provision for applying package attributes to ‘cloned’ copies of subroutines used as closures. (See [Making References in perlref](#) for information on closures.)

Package-specific attribute handling may change incompatibly in a future release.

When an attribute list is present in a declaration, a check is made to see whether an attribute ‘modify’ handler is present in the appropriate package (or its @ISA inheritance tree). Similarly, when `attributes::get` is called on a valid reference, a check is made for an appropriate attribute ‘fetch’ handler. See ["EXAMPLES"](#) to see how the "appropriate package" determination works.

The handler names are based on the underlying type of the variable being declared or of the reference passed. Because these attributes are associated with subroutine or variable declarations, this deliberately ignores any possibility of being blessed into some package. Thus, a subroutine declaration uses "CODE" as its *type*, and even a blessed hash reference uses "HASH" as its *type*.

The class methods invoked for modifying and fetching are these:

FETCH_type_ATTRIBUTES

This method receives a single argument, which is a reference to the variable or subroutine for which package-defined attributes are desired. The expected return value is a list of associated attributes. This list may be empty.

MODIFY_type_ATTRIBUTES

This method is called with two fixed arguments, followed by the list of attributes from the relevant declaration. The two fixed arguments are the relevant package name and a reference to the declared subroutine or variable. The expected return value is a list of attributes which were not recognized by this handler. Note that this allows for a derived class to delegate a call to its base class, and then only

examine the attributes which the base class didn't already handle for it.

The call to this method is currently made *during* the processing of the declaration. In particular, this means that a subroutine reference will probably be for an undefined subroutine, even if this declaration is actually part of the definition.

Calling `attributes::get()` from within the scope of a null package declaration `package ;` for an unblessed variable reference will not provide any starting package name for the 'fetch' method lookup. Thus, this circumstance will not result in a method call for package-defined attributes. A named subroutine knows to which symbol table entry it belongs (or originally belonged), and it will use the corresponding package. An anonymous subroutine knows the package name into which it was compiled (unless it was also compiled with a null package declaration), and so it will use that package name.

Syntax of Attribute Lists

An attribute list is a sequence of attribute specifications, separated by whitespace or a colon (with optional whitespace). Each attribute specification is a simple name, optionally followed by a parenthesised parameter list. If such a parameter list is present, it is scanned past as for the rules for the `q()` operator. (See [Quote and Quote-like Operators in perlop](#).) The parameter list is passed as it was found, however, and not as `per q()`.

Some examples of syntactically valid attribute lists:

```
switch(10,foo(7,3))    : expensive
Ugly('\(') :Bad
_5x5
locked method
```

Some examples of syntactically invalid attribute lists (with annotation):

```
switch(10,foo())      # ()-string not balanced
Ugly('(')             # ()-string not balanced
5x5                   # "5x5" not a valid identifier
Y2::north             # "Y2::north" not a simple identifier
foo + bar             # "+" neither a colon nor whitespace
```

EXPORTS

Default exports

None.

Available exports

The routines `get` and `reftype` are exportable.

Export tags defined

The `:ALL` tag will get all of the above exports.

EXAMPLES

Here are some samples of syntactically valid declarations, with annotation as to how they resolve internally into use `attributes` invocations by `perl`. These examples are primarily useful to see how the "appropriate package" is found for the possible method lookups for package-defined attributes.

1. Code:

```
package Canine;
package Dog;
my Canine $spot : Watchful ;
```

Effect:

```
use attributes ();
attributes::->import(Canine => \$spot, "Watchful");
```

2. Code:

```
package Felis;  
my $cat : Nervous;
```

Effect:

```
use attributes ();  
attributes::->import(Felis => \$cat, "Nervous");
```

3. Code:

```
package X;  
sub foo : locked ;
```

Effect:

```
use attributes X => \&foo, "locked";
```

4. Code:

```
package X;  
sub Y::x : locked { 1 }
```

Effect:

```
use attributes Y => \&Y::x, "locked";
```

5. Code:

```
package X;  
sub foo { 1 }  
  
package Y;  
BEGIN { *bar = \&X::foo; }  
  
package Z;  
sub Y::bar : locked ;
```

Effect:

```
use attributes X => \&X::foo, "locked";
```

This last example is purely for purposes of completeness. You should not be trying to mess with the attributes of something in a package that's not your own.

SEE ALSO

"Private Variables via my ()" and *Subroutine Attributes in perlsb* for details on the basic declarations; *attrs* for the obsolescent form of subroutine attribute specification which this module replaces; *use* for details on the normal invocation mechanism.

NAME

attrs – set/get attributes of a subroutine (deprecated)

SYNOPSIS

```
sub foo {  
    use attrs qw(locked method);  
    ...  
}  
  
@a = attrs::get(\&foo);
```

DESCRIPTION

NOTE: Use of this pragma is deprecated. Use the syntax

```
sub foo : locked method { }
```

to declare attributes instead. See also [attributes](#).

This pragma lets you set and get attributes for subroutines. Setting attributes takes place at compile time; trying to set invalid attribute names causes a compile-time error. Calling `attrs::get` on a subroutine reference or name returns its list of attribute names. Notice that `attrs::get` is not exported. Valid attributes are as follows.

method

Indicates that the invoking subroutine is a method.

locked

Setting this attribute is only meaningful when the subroutine or method is to be called by multiple threads. When set on a method subroutine (i.e. one marked with the **method** attribute above), perl ensures that any invocation of it implicitly locks its first argument before execution. When set on a non-method subroutine, perl ensures that a lock is taken on the subroutine itself before execution. The semantics of the lock are exactly those of one explicitly taken with the `lock` operator immediately after the subroutine is entered.

NAME

AutoLoader – load subroutines only on demand

SYNOPSIS

```
package Foo;
use AutoLoader 'AUTOLOAD';    # import the default AUTOLOAD subroutine

package Bar;
use AutoLoader;                # don't import AUTOLOAD, define our own
sub AUTOLOAD {
    ...
    $AutoLoader::AUTOLOAD = "...";
    goto &AutoLoader::AUTOLOAD;
}
```

DESCRIPTION

The **AutoLoader** module works with the **AutoSplit** module and the `__END__` token to defer the loading of some subroutines until they are used rather than loading them all at once.

To use **AutoLoader**, the author of a module has to place the definitions of subroutines to be autoloaded after an `__END__` token. (See [perldata](#).) The **AutoSplit** module can then be run manually to extract the definitions into individual files *auto/funcname.al*.

AutoLoader implements an AUTOLOAD subroutine. When an undefined subroutine is called in a client module of **AutoLoader**, **AutoLoader**'s AUTOLOAD subroutine attempts to locate the subroutine in a file with a name related to the location of the file from which the client module was read. As an example, if *POSIX.pm* is located in */usr/local/lib/perl5/POSIX.pm*, **AutoLoader** will look for perl subroutines **POSIX** in */usr/local/lib/perl5/auto/POSIX/*.al*, where the *.al* file has the same name as the subroutine, sans package. If such a file exists, AUTOLOAD will read and evaluate it, thus (presumably) defining the needed subroutine. AUTOLOAD will then `goto` the newly defined subroutine.

Once this process completes for a given function, it is defined, so future calls to the subroutine will bypass the AUTOLOAD mechanism.

Subroutine Stubs

In order for object method lookup and/or prototype checking to operate correctly even when methods have not yet been defined it is necessary to "forward declare" each subroutine (as in `sub NAME;`). See [SYNOPSIS in perlsub](#). Such forward declaration creates "subroutine stubs", which are place holders with no code.

The **AutoSplit** and **AutoLoader** modules automate the creation of forward declarations. The **AutoSplit** module creates an 'index' file containing forward declarations of all the **AutoSplit** subroutines. When the **AutoLoader** module is 'use'd it loads these declarations into its callers package.

Because of this mechanism it is important that **AutoLoader** is always used and not required.

Using AutoLoader's AUTOLOAD Subroutine

In order to use **AutoLoader**'s AUTOLOAD subroutine you *must* explicitly import it:

```
use AutoLoader 'AUTOLOAD';
```

Overriding AutoLoader's AUTOLOAD Subroutine

Some modules, mainly extensions, provide their own AUTOLOAD subroutines. They typically need to check for some special cases (such as constants) and then fallback to **AutoLoader**'s AUTOLOAD for the rest.

Such modules should *not* import **AutoLoader**'s AUTOLOAD subroutine. Instead, they should define their own AUTOLOAD subroutines along these lines:

```

use AutoLoader;
use Carp;

sub AUTOLOAD {
    my $sub = $AUTOLOAD;
    (my $constname = $sub) =~ s/.*:://;
    my $val = constant($constname, @_ ? $_[0] : 0);
    if ($! != 0) {
        if ($! =~ /Invalid/ || ${!{EINVAL}}) {
            $AutoLoader::AUTOLOAD = $sub;
            goto &AutoLoader::AUTOLOAD;
        }
        else {
            croak "Your vendor has not defined constant $constname";
        }
    }
    *$sub = sub { $val }; # same as: eval "sub $sub { $val }";
    goto &$sub;
}

```

If any module's own AUTOLOAD subroutine has no need to fallback to the AutoLoader's AUTOLOAD subroutine (because it doesn't have any AutoSplit subroutines), then that module should not use **AutoLoader** at all.

Package Lexicals

Package lexicals declared with `my` in the main block of a package using **AutoLoader** will not be visible to auto-loaded subroutines, due to the fact that the given scope ends at the `__END__` marker. A module using such variables as package globals will not work properly under the **AutoLoader**.

The `vars` pragma (see [vars in perlmod](#)) may be used in such situations as an alternative to explicitly qualifying all globals with the package namespace. Variables pre-declared with this pragma will be visible to any autoloading routines (but will not be invisible outside the package, unfortunately).

Not Using AutoLoader

You can stop using AutoLoader by simply

```
no AutoLoader;
```

AutoLoader vs. SelfLoader

The **AutoLoader** is similar in purpose to **SelfLoader**: both delay the loading of subroutines.

SelfLoader uses the `__DATA__` marker rather than `__END__`. While this avoids the use of a hierarchy of disk files and the associated open/close for each routine loaded, **SelfLoader** suffers a startup speed disadvantage in the one-time parsing of the lines after `__DATA__`, after which routines are cached. **SelfLoader** can also handle multiple packages in a file.

AutoLoader only reads code as it is requested, and in many cases should be faster, but requires a mechanism like **AutoSplit** be used to create the individual files. [ExtUtils::MakeMaker](#) will invoke **AutoSplit** automatically if **AutoLoader** is used in a module source file.

CAVEATS

AutoLoaders prior to Perl 5.002 had a slightly different interface. Any old modules which use **AutoLoader** should be changed to the new calling style. Typically this just means changing a `require` to a `use`, adding the explicit `'AUTOLOAD'` import if needed, and removing **AutoLoader** from `@ISA`.

On systems with restrictions on file name length, the file corresponding to a subroutine may have a shorter name than the routine itself. This can lead to conflicting file names. The *AutoSplit* package warns of these potential conflicts when used to split a module.

AutoLoader may fail to find the autosplit files (or even find the wrong ones) in cases where @INC contains relative paths, **and** the program does `chdir`.

SEE ALSO

[SelfLoader](#) – an autoloader that doesn't use external files.

NAME

AutoSplit – split a package for autoloading

SYNOPSIS

```
autosplit($file, $dir, $keep, $check, $modtime);  
autosplit_lib_modules(@modules);
```

DESCRIPTION

This function will split up your program into files that the AutoLoader module can handle. It is used by both the standard perl libraries and by the MakeMaker utility, to automatically configure libraries for autoloading.

The `autosplit` interface splits the specified file into a hierarchy rooted at the directory `$dir`. It creates directories as needed to reflect class hierarchy, and creates the file ***autosplit.ix***. This file acts as both forward declaration of all package routines, and as timestamp for the last update of the hierarchy.

The remaining three arguments to `autosplit` govern other options to the autosplitter.

`$keep`

If the third argument, `$keep`, is false, then any pre-existing `*.al` files in the autoload directory are removed if they are no longer part of the module (obsoleted functions). `$keep` defaults to 0.

`$check`

The fourth argument, `$check`, instructs `autosplit` to check the module currently being split to ensure that it includes a use specification for the AutoLoader module, and skips the module if AutoLoader is not detected. `$check` defaults to 1.

`$modtime`

Lastly, the `$modtime` argument specifies that `autosplit` is to check the modification time of the module against that of the `autosplit.ix` file, and only split the module if it is newer. `$modtime` defaults to 1.

Typical use of AutoSplit in the perl MakeMaker utility is via the command-line with:

```
perl -e 'use AutoSplit; autosplit($ARGV[0], $ARGV[1], 0, 1, 1)'
```

Defined as a Make macro, it is invoked with file and directory arguments; `autosplit` will split the specified file into the specified directory and delete obsolete `.al` files, after checking first that the module does use the AutoLoader, and ensuring that the module is not already currently split in its current form (the modtime test).

The `autosplit_lib_modules` form is used in the building of perl. It takes as input a list of files (modules) that are assumed to reside in a directory **lib** relative to the current directory. Each file is sent to the autosplitter one at a time, to be split into the directory **lib/auto**.

In both usages of the autosplitter, only subroutines defined following the perl `__END__` token are split out into separate files. Some routines may be placed prior to this marker to force their immediate loading and parsing.

Multiple packages

As of version 1.01 of the AutoSplit module it is possible to have multiple packages within a single file. Both of the following cases are supported:

```
package NAME;  
__END__  
sub AAA { ... }  
package NAME::option1;  
sub BBB { ... }  
package NAME::option2;
```

```
sub BBB { ... }  
  
package NAME;  
__END__  
sub AAA { ... }  
sub NAME::option1::BBB { ... }  
sub NAME::option2::BBB { ... }
```

DIAGNOSTICS

AutoSplit will inform the user if it is necessary to create the top-level directory specified in the invocation. It is preferred that the script or installation process that invokes AutoSplit have created the full directory path ahead of time. This warning may indicate that the module is being split into an incorrect path.

AutoSplit will warn the user of all subroutines whose name causes potential file naming conflicts on machines with drastically limited (8 characters or less) file name length. Since the subroutine name is used as the file name, these warnings can aid in portability to such systems.

Warnings are issued and the file skipped if AutoSplit cannot locate either the `__END__` marker or a "package Name;"-style specification.

AutoSplit will also emit general diagnostics for inability to create directories or files.

NAME

autouse – postpone load of modules until a function is used

SYNOPSIS

```
use autouse 'Carp' => qw(carp croak);
carp "this carp was predeclared and autoused ";
```

DESCRIPTION

If the module `Module` is already loaded, then the declaration

```
use autouse 'Module' => qw(func1 func2($;$));
```

is equivalent to

```
use Module qw(func1 func2);
```

if `Module` defines `func2()` with prototype `($;$)`, and `func1()` has no prototypes. (At least if `Module` uses `Exporter`'s `import`, otherwise it is a fatal error.)

If the module `Module` is not loaded yet, then the above declaration declares functions `func1()` and `func2()` in the current package. When these functions are called, they load the package `Module` if needed, and substitute themselves with the correct definitions.

`=begin _deprecated`

```
use Module qw(Module::func3);
```

will work and is the equivalent to:

```
use Module qw(func3);
```

It is not a very useful feature and has been deprecated.

`=end _deprecated`

WARNING

Using `autouse` will move important steps of your program's execution from compile time to runtime. This can

- Break the execution of your program if the module you autoused has some initialization which it expects to be done early.
- hide bugs in your code since important checks (like correctness of prototypes) is moved from compile time to runtime. In particular, if the prototype you specified on `autouse` line is wrong, you will not find it out until the corresponding function is executed. This will be very unfortunate for functions which are not always called (note that for such functions `autouseing` gives biggest win, for a workaround see below).

To alleviate the second problem (partially) it is advised to write your scripts like this:

```
use Module;
use autouse Module => qw(carp($) croak(&$));
carp "this carp was predeclared and autoused ";
```

The first line ensures that the errors in your argument specification are found early. When you ship your application you should comment out the first line, since it makes the second one useless.

AUTHOR

Ilya Zakharevich (ilya@math.ohio-state.edu)

SEE ALSO

`perl(1)`.

NAME

B – The Perl Compiler

SYNOPSIS

```
use B;
```

DESCRIPTION

The B module supplies classes which allow a Perl program to delve into its own innards. It is the module used to implement the "backends" of the Perl compiler. Usage of the compiler does not require knowledge of this module: see the *O* module for the user-visible part. The B module is of use to those who want to write new compiler backends. This documentation assumes that the reader knows a fair amount about perl's internals including such things as SVs, OPs and the internal symbol table and syntax tree of a program.

OVERVIEW OF CLASSES

The C structures used by Perl's internals to hold SV and OP information (PVIV, AV, HV, ..., OP, SVOP, UNOP, ...) are modelled on a class hierarchy and the B module gives access to them via a true object hierarchy. Structure fields which point to other objects (whether types of SV or types of OP) are represented by the B module as Perl objects of the appropriate class. The bulk of the B module is the methods for accessing fields of these structures. Note that all access is read-only: you cannot modify the internals by using this module.

SV-RELATED CLASSES

B::IV, B::NV, B::RV, B::PV, B::PVIV, B::PVNV, B::PVMG, B::BM, B::PVLV, B::AV, B::HV, B::CV, B::GV, B::FM, B::IO. These classes correspond in the obvious way to the underlying C structures of similar names. The inheritance hierarchy mimics the underlying C "inheritance". Access methods correspond to the underlying C macros for field access, usually with the leading "class indication" prefix removed (Sv, Av, Hv, ...). The leading prefix is only left in cases where its removal would cause a clash in method name. For example, GvREFCNT stays as-is since its abbreviation would clash with the "superclass" method REFCNT (corresponding to the C function SvREFCNT).

B::SV METHODS

REFCNT
FLAGS

B::IV METHODS

IV Returns the value of the IV, *interpreted as a signed integer*. This will be misleading if FLAGS & SVf_IVisUV. Perhaps you want the int_value method instead?

IVX
UVX
int_value

This method returns the value of the IV as an integer. It differs from IV in that it returns the correct value regardless of whether it's stored signed or unsigned.

needs64bits
packiv

B::NV METHODS

NV
NVX

B::RV METHODS

RV

B::PV METHODS

PV This method is the one you usually want. It constructs a string using the length and offset information in the struct: for ordinary scalars it will return the string that you'd see from Perl, even if it contains null characters.

RV Same as B::RV::RV, except that it will `die()` if the PV isn't a reference.

PVX

This method is less often useful. It assumes that the string stored in the struct is null-terminated, and disregards the length information.

It is the appropriate method to use if you need to get the name of a lexical variable from a padname array. Lexical variable names are always stored with a null terminator, and the length field (SvCUR) is overloaded for other purposes and can't be relied on here.

B::PVMG METHODS

MAGIC

SvSTASH

B::MAGIC METHODS

MOREMAGIC

precomp

Only valid on r-magic, returns the string that generated the regexp.

PRIVATE

TYPE

FLAGS

OBJ

Will `die()` if called on r-magic.

PTR

REGEX

Only valid on r-magic, returns the integer value of the REGEX stored in the MAGIC.

B::PVLV METHODS

TARGOFF

TARGLEN

TYPE

TARG

B::BM METHODS

USEFUL

PREVIOUS

RARE

TABLE

B::GV METHODS

is_empty

This method returns TRUE if the GP field of the GV is NULL.

NAME

SAFENAME

This method returns the name of the glob, but if the first character of the name is a control character, then it converts it to ^X first, so that *^G would return "^G" rather than "\cG".

It's useful if you want to print out the name of a variable. If you restrict yourself to globs which exist at compile-time then the result ought to be unambiguous, because code like `{ ^G } = 1` is compiled as two ops – a constant string and a dereference (`rv2gv`) – so that the glob is created at runtime.

If you're working with globs at runtime, and need to disambiguate `*^G` from `{ ^G }`, then you should use the `raw NAME` method.

STASH
SV
IO
FORM
AV
HV
EGV
CV
CVGEN
LINE
FILE
FILEGV
GvREFCNT
FLAGS

B::IO METHODS

LINES
PAGE
PAGE_LEN
LINES_LEFT
TOP_NAME
TOP_GV
FMT_NAME
FMT_GV
BOTTOM_NAME
BOTTOM_GV
SUBPROCESS
IoTYPE
IoFLAGS
IsSTD

Takes one arguments (`'stdin'` | `'stdout'` | `'stderr'`) and returns true if the IoIFP of the object is equal to the handle whose name was passed as argument (i.e. `$io-IsSTD('stderr')` is true if `IoIFP($io) == PerlIO_stdin()`).

B::AV METHODS

FILL
MAX
OFF
ARRAY
AvFLAGS

B::CV METHODS

STASH
START
ROOT
GV
FILE

DEPTH
PADLIST
OUTSIDE
XSUB
XSUBANY

For constant subroutines, returns the constant SV returned by the subroutine.

CvFLAGS
const_sv

B::HV METHODS

FILL
MAX
KEYS
RITER
NAME
PMROOT
ARRAY

OP-RELATED CLASSES

B::OP, B::UNOP, B::BINOP, B::LOGOP, B::LISTOP, B::PMOP, B::SVOP, B::PADOP, B::PVOP, B::CVOP, B::LOOP, B::COP. These classes correspond in the obvious way to the underlying C structures of similar names. The inheritance hierarchy mimics the underlying C "inheritance". Access methods correspond to the underlying C structre field names, with the leading "class indication" prefix removed (op_).

B::OP METHODS

next
sibling
name

This returns the op name as a string (e.g. "add", "rv2av").

ppaddr

This returns the function name as a string (e.g. "PL_ppaddr[OP_ADD]", "PL_ppaddr[OP_RV2AV]").

desc

This returns the op description from the global C PL_op_desc array (e.g. "addition" "array deref").

targ
type
seq
flags
private

B::UNOP METHOD

first

B::BINOP METHOD

last

B::LOGOP METHOD

other

B::LISTOP METHOD

children

B::PMOP METHODS

pmreplroot
pmreplstart
pmnext
pmregexp
pmflags
pmdynflags
pmpermflags
precomp
pmoffet

Only when perl was compiled with ithreads.

B::SVOP METHOD

sv
gv

B::PADOP METHOD

padix

B::PVOP METHOD

pv

B::LOOP METHODS

redoop
nextop
lastop

B::COP METHODS

label
stash
file
cop_seq
arybase
line

FUNCTIONS EXPORTED BY B

The B module exports a variety of functions: some are simple utility functions, others provide a Perl program with a way to get an initial "handle" on an internal object.

main_cv

Return the (faked) CV corresponding to the main part of the Perl program.

init_av

Returns the AV object (i.e. in class B::AV) representing INIT blocks.

begin_av

Returns the AV object (i.e. in class B::AV) representing BEGIN blocks.

end_av

Returns the AV object (i.e. in class B::AV) representing END blocks.

main_root

Returns the root op (i.e. an object in the appropriate B::OP-derived class) of the main part of the Perl program.

main_start

Returns the starting op of the main part of the Perl program.

comppadlist

Returns the AV object (i.e. in class B::AV) of the global comppadlist.

regex_padav

Only when perl was compiled with ithreads.

sv_undef

Returns the SV object corresponding to the C variable `sv_undef`.

sv_yes

Returns the SV object corresponding to the C variable `sv_yes`.

sv_no

Returns the SV object corresponding to the C variable `sv_no`.

amagic_generation

Returns the SV object corresponding to the C variable `amagic_generation`.

walkoptree(OP, METHOD)

Does a tree-walk of the syntax tree based at OP and calls METHOD on each op it visits. Each node is visited before its children. If `walkoptree_debug` (q.v.) has been called to turn debugging on then the method `walkoptree_debug` is called on each op before METHOD is called.

walkoptree_debug(DEBUG)

Returns the current debugging flag for `walkoptree`. If the optional DEBUG argument is non-zero, it sets the debugging flag to that. See the description of `walkoptree` above for what the debugging flag does.

walksymtable(SYMREF, METHOD, RECURSE, PREFIX)

Walk the symbol table starting at SYMREF and call METHOD on each symbol (a B::GV object) visited. When the walk reaches package symbols (such as "Foo::") it invokes RECURSE, passing in the symbol name, and only recurses into the package if that sub returns true.

PREFIX is the name of the SYMREF you're walking.

For example...

```
# Walk CGI's symbol table calling print_subs on each symbol.
# Only recurse into CGI::Util::
walksymtable(\%CGI::, 'print_subs', sub { $_[0] eq 'CGI::Util::' },
             'CGI::');
```

`print_subs()` is a B::GV method you have declared.

svref_2object(SV)

Takes any Perl variable and turns it into an object in the appropriate B::OP-derived or B::SV-derived class. Apart from functions such as `main_root`, this is the primary way to get an initial "handle" on an internal perl data structure which can then be followed with the other access methods.

ppname(OPNUM)

Return the PP function name (e.g. "pp_add") of op number OPNUM.

hash(STR)

Returns a string in the form "0x..." representing the value of the internal hash function used by perl on string STR.

`cast_I32(I)`

Casts `I` to the internal I32 type used by that perl.

`minus_c`

Does the equivalent of the `-c` command-line option. Obviously, this is only useful in a `BEGIN` block or else the flag is set too late.

`cstring(STR)`

Returns a double-quote-surrounded escaped version of `STR` which can be used as a string in C source code.

`perlstring(STR)`

Returns a double-quote-surrounded escaped version of `STR` which can be used as a string in Perl source code.

`class(OBJ)`

Returns the class of an object without the part of the classname preceding the first `::`. This is used to turn `"B::UNOP"` into `"UNOP"` for example.

`threadsv_names`

In a perl compiled for threads, this returns a list of the special per-thread threadsv variables.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

B::Asmdata – Autogenerated data about Perl ops, used to generate bytecode

SYNOPSIS

```
use B::Asmdata qw(%insn_data @insn_name @optype @specialsv_name);
```

DESCRIPTION

Provides information about Perl ops in order to generate bytecode via a bunch of exported variables. Its mostly used by B::Assembler and B::Disassembler.

%insn_data

```
my($bytecode_num, $put_sub, $get_meth) = @{$insn_data{$op_name}};
```

For a given \$op_name (for example, 'cop_label', 'sv_flags', etc...) you get an array ref containing the bytecode number of the op, a reference to the subroutine used to 'PUT', and the name of the method used to 'GET'.

=for _private Add more detail about what \$put_sub and \$get_meth are and how to use them.

@insn_name

```
my $op_name = $insn_name[$bytecode_num];
```

A simple mapping of the bytecode number to the name of the op. Suitable for using with %insn_data like so:

```
my $op_info = $insn_data{$insn_name[$bytecode_num]};
```

@optype

```
my $op_type = $optype[$op_type_num];
```

A simple mapping of the op type number to its type (like 'COP' or 'BINOP').

@specialsv_name

```
my $sv_name = $specialsv_name[$sv_index];
```

Certain SV types are considered 'special'. They're represented by B::SPECIAL and are referred to by a number from the specialsv_list. This array maps that number back to the name of the SV (like 'Nullsv' or '&PL_sv_undef').

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

B::Assembler – Assemble Perl bytecode

SYNOPSIS

```
use B::Assembler qw(newasm endasm assemble);
newasm(\&printsub);      # sets up for assembly
assemble($buf);          # assembles one line
endasm();                 # closes down

use B::Assembler qw(assemble_fh);
assemble_fh($fh, \&printsub); # assemble everything in $fh
```

DESCRIPTION

See *ext/B/B/Assembler.pm*.

AUTHORS

Malcolm Beattie, mbeattie@sable.ox.ac.uk Per-statement interface by Benjamin Stuhl,
sho_pi@hotmail.com

NAME

B::Bblock – Walk basic blocks

SYNOPSIS

```
# External interface
perl -MO=Bblock[,OPTIONS] foo.pl

# Programmatic API
use B::Bblock qw(find_leaders);
my $leaders = find_leaders($root_op, $start_op);
```

DESCRIPTION

This module is used by the B::CC back end. It walks "basic blocks". A basic block is a series of operations which is known to execute from start to finish, with no possibility of branching or halting.

It can be used either stand alone or from inside another program.

=for _private Somebody who understands the stand-alone options document them, please.

Functions**find_leaders**

```
my $leaders = find_leaders($root_op, $start_op);
```

Given the root of the op tree and an op from which to start processing, it will return a hash ref representing all the ops which start a block.

=for _private The above description may be somewhat wrong.

The values of %\$leaders are the op objects themselves. Keys are \$\$op addresses.

=for _private Above cribbed from B::CC's comments. What's a \$\$op address?

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

B::Bytecode – Perl compiler's bytecode backend

SYNOPSIS

```
perl -MO=Bytecode[,OPTIONS] foo.pl
```

DESCRIPTION

This compiler backend takes Perl source and generates a platform-independent bytecode encapsulating code to load the internal structures perl uses to run your program. When the generated bytecode is loaded in, your program is ready to run, reducing the time which perl would have taken to load and parse your program into its internal semi-compiled form. That means that compiling with this backend will not help improve the runtime execution speed of your program but may improve the start-up time. Depending on the environment in which your program runs this may or may not be a help.

The resulting bytecode can be run with a special byteperl executable or (for non-main programs) be loaded via the `byteload_fh` function in the **B** module.

OPTIONS

If there are any non-option arguments, they are taken to be names of objects to be saved (probably doesn't work properly yet). Without extra arguments, it saves the main program.

-ofilename

Output to filename instead of STDOUT.

-afilename

Append output to filename.

— Force end of options.

-f Force optimisations on or off one at a time. Each can be preceded by **no-** to turn the option off (e.g. **-fno-compress-nullops**).

-fcompress-nullops

Only fills in the necessary fields of ops which have been optimised away by perl's internal compiler.

-fomit-sequence-numbers

Leaves out code to fill in the `op_seq` field of all ops which is only used by perl's internal compiler.

-fbypass-nullops

If `op-op_next` ever points to a NULLOP, replaces the `op_next` field with the first non-NULLOP in the path of execution.

-On

Optimisation level ($n = 0, 1, 2, \dots$). **-O** means **-O1**. **-O1** sets **-fcompress-nullops -fomit-sequence numbers**. **-O2** adds **-fbypass-nullops**.

-D Debug options (concatenated or separate flags like `perl -D`).

-Do Prints each OP as it's processed.

-Db Print debugging information about bytecompiler progress.

-Da Tells the (bytecode) assembler to include source assembler lines in its output as bytecode comments.

-DC

Prints each CV taken from the final symbol tree walk.

-S Output (bytecode) assembler source rather than piping it through the assembler and outputting bytecode.

-upackage

Stores package in the output.

EXAMPLES

```
perl -MO=Bytecode,-O6,-ofoo.plc,-umain foo.pl
```

```
perl -MO=Bytecode,-S,-umain foo.pl > foo.S
```

```
assemble foo.S > foo.plc
```

Note that assemble lives in the B subdirectory of your perl library directory. The utility called perlcc may also be used to help make use of this compiler.

```
perl -MO=Bytecode,-uFoo,-oFoo.pmc Foo.pm
```

BUGS

Output is still huge and there are still occasional crashes during either compilation or ByteLoading. Current status: experimental.

AUTHORS

Malcolm Beattie, mbeattie@sable.ox.ac.uk Benjamin Stuhl, sho_pi@hotmail.com

NAME

B::C – Perl compiler's C backend

SYNOPSIS

```
perl -MO=C[,OPTIONS] foo.pl
```

DESCRIPTION

This compiler backend takes Perl source and generates C source code corresponding to the internal structures that perl uses to run your program. When the generated C source is compiled and run, it cuts out the time which perl would have taken to load and parse your program into its internal semi-compiled form. That means that compiling with this backend will not help improve the runtime execution speed of your program but may improve the start-up time. Depending on the environment in which your program runs this may be either a help or a hindrance.

OPTIONS

If there are any non-option arguments, they are taken to be names of objects to be saved (probably doesn't work properly yet). Without extra arguments, it saves the main program.

-ofilename

Output to filename instead of STDOUT

-v Verbose compilation (currently gives a few compilation statistics).

— Force end of options

-uPackname

Force apparently unused subs from package Packname to be compiled. This allows programs to use `eval "foo()"` even when sub `foo` is never seen to be used at compile time. The down side is that any subs which really are never used also have code generated. This option is necessary, for example, if you have a signal handler `foo` which you initialise with `$SIG{BAR} = "foo"`. A better fix, though, is just to change it to `$SIG{BAR} = \&foo`. You can have multiple **-u** options. The compiler tries to figure out which packages may possibly have subs in which need compiling but the current version doesn't do it very well. In particular, it is confused by nested packages (i.e. of the form `A: :B`) where package `A` does not contain any subs.

-D Debug options (concatenated or separate flags like `perl -D`).

-Do OPs, prints each OP as it's processed

-Dc COPs, prints COPs as processed (incl. file & line num)

-DA

prints AV information on saving

-DC

prints CV information on saving

-DM

prints MAGIC information on saving

-f Force options/optimisations on or off one at a time. You can explicitly disable an option using **-fno-option**. All options default to **disabled**.

-fcog

Copy-on-grow: PVs declared and initialised statically.

-fsave-data

Save package::DATA filehandles (only available with PerlIO).

-fppaddr

Optimize the initialization of `op_ppaddr`.

-fwarn-sv

Optimize the initialization of `cop_warnings`.

-fuse-script-name

Use the script name instead of the program name as `$0`.

-fsave-sig-hash

Save compile-time modifications to the `%SIG` hash.

-On

Optimisation level ($n = 0, 1, 2, \dots$). **-O** means **-O1**.

-O0 Disable all optimizations.

-O1 Enable **-fcog**.

-O2 Enable **-fppaddr**, **-fwarn-sv**.

-llimit

Some C compilers impose an arbitrary limit on the length of string constants (e.g. 2048 characters for Microsoft Visual C++). The **-llimit** options tells the C backend not to generate string literals exceeding that limit.

EXAMPLES

```
perl -MO=C,-ofoo.c foo.pl
perl cc_harness -o foo foo.c
```

Note that `cc_harness` lives in the `B` subdirectory of your perl library directory. The utility called `perlcc` may also be used to help make use of this compiler.

```
perl -MO=C,-v,-DcA,-l2048 bar.pl > /dev/null
```

BUGS

Plenty. Current status: experimental.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

B::CC – Perl compiler's optimized C translation backend

SYNOPSIS

```
perl -MO=CC[,OPTIONS] foo.pl
```

DESCRIPTION

This compiler backend takes Perl source and generates C source code corresponding to the flow of your program. In other words, this backend is somewhat a "real" compiler in the sense that many people think about compilers. Note however that, currently, it is a very poor compiler in that although it generates (mostly, or at least sometimes) correct code, it performs relatively few optimisations. This will change as the compiler develops. The result is that running an executable compiled with this backend may start up more quickly than running the original Perl program (a feature shared by the **C** compiler backend—see **B::C**) and may also execute slightly faster. This is by no means a good optimising compiler—yet.

OPTIONS

If there are any non-option arguments, they are taken to be names of objects to be saved (probably doesn't work properly yet). Without extra arguments, it saves the main program.

-ofilename

Output to filename instead of STDOUT

-v Verbose compilation (currently gives a few compilation statistics).

— Force end of options

-uPackname

Force apparently unused subs from package Packname to be compiled. This allows programs to use `eval "foo()"` even when sub `foo` is never seen to be used at compile time. The down side is that any subs which really are never used also have code generated. This option is necessary, for example, if you have a signal handler `foo` which you initialise with `$SIG{BAR} = "foo"`. A better fix, though, is just to change it to `$SIG{BAR} = \&foo`. You can have multiple **-u** options. The compiler tries to figure out which packages may possibly have subs in which need compiling but the current version doesn't do it very well. In particular, it is confused by nested packages (i.e. of the form `A: :B`) where package `A` does not contain any subs.

-mModulename

Instead of generating source for a runnable executable, generate source for an XSUB module. The `boot_Modulename` function (which DynaLoader can look for) does the appropriate initialisation and runs the main part of the Perl source that is being compiled.

-D Debug options (concatenated or separate flags like `perl -D`).

-Dr Writes debugging output to STDERR just as it's about to write to the program's runtime (otherwise writes debugging info as comments in its C output).

-DO

Outputs each OP as it's compiled

-Ds Outputs the contents of the shadow stack at each OP

-Dp Outputs the contents of the shadow pad of lexicals as it's loaded for each sub or the main program.

-Dq Outputs the name of each fake PP function in the queue as it's about to process it.

-DI Output the filename and line number of each original line of Perl code as it's processed (`pp_nextstate`).

-Dt Outputs timing information of compilation stages.

-f Force optimisations on or off one at a time.

-ffreetmps-each-bblock

Delays FREETMPS from the end of each statement to the end of the each basic block.

-ffreetmps-each-loop

Delays FREETMPS from the end of each statement to the end of the group of basic blocks forming a loop. At most one of the freetmps-each-* options can be used.

-fomit-taint

Omits generating code for handling perl's tainting mechanism.

-On

Optimisation level (n = 0, 1, 2, ...). **-O** means **-O1**. Currently, **-O1** sets **-ffreetmps-each-bblock** and **-O2** sets **-ffreetmps-each-loop**.

EXAMPLES

```
perl -MO=CC,-O2,-ofoo.c foo.pl
perl cc_harness -o foo foo.c
```

Note that `cc_harness` lives in the B subdirectory of your perl library directory. The utility called `perlcc` may also be used to help make use of this compiler.

```
perl -MO=CC,-mFoo,-oFoo.c Foo.pm
perl cc_harness -shared -c -o Foo.so Foo.c
```

BUGS

Plenty. Current status: experimental.

DIFFERENCES

These aren't really bugs but they are constructs which are heavily tied to perl's compile-and-go implementation and with which this compiler backend cannot cope.

Loops

Standard perl calculates the target of "next", "last", and "redo" at run-time. The compiler calculates the targets at compile-time. For example, the program

```
sub skip_on_odd { next NUMBER if $_[0] % 2 }
NUMBER: for ($i = 0; $i < 5; $i++) {
    skip_on_odd($i);
    print $i;
}
```

produces the output

```
024
```

with standard perl but gives a compile-time error with the compiler.

Context of ".."

The context (scalar or array) of the ".." operator determines whether it behaves as a range or a flip/flop. Standard perl delays until runtime the decision of which context it is in but the compiler needs to know the context at compile-time. For example,

```
@a = (4,6,1,0,0,1);
sub range { (shift @a)..(shift @a) }
print range();
while (@a) { print scalar(range()) }
```

generates the output

```
456123E0
```

with standard Perl but gives a compile-time error with compiled Perl.

Arithmetic

Compiled Perl programs use native C arithmetic much more frequently than standard perl. Operations on large numbers or on boundary cases may produce different behaviour.

Deprecated features

Features of standard perl such as `$[` which have been deprecated in standard perl since Perl5 was released have not been implemented in the compiler.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

B::Concise – Walk Perl syntax tree, printing concise info about ops

SYNOPSIS

```
perl -MO=Concise[,OPTIONS] foo.pl

use B::Concise qw(set_style add_callback);
```

DESCRIPTION

This compiler backend prints the internal OPs of a Perl program's syntax tree in one of several space-efficient text formats suitable for debugging the inner workings of perl or other compiler backends. It can print OPs in the order they appear in the OP tree, in the order they will execute, or in a text approximation to their tree structure, and the format of the information displayed is customizable. Its function is similar to that of perl's **-Dx** debugging flag or the **B::Terse** module, but it is more sophisticated and flexible.

EXAMPLE

Here's is a short example of output, using the default formatting conventions :

```
% perl -MO=Concise -e '$a = $b + 42'
8  <@> leave[t1] vKP/REFC ->(end)
1   <0> enter ->2
2   <;> nextstate(main 1 -e:1) v ->3
7   <2> sassign vKS/2 ->8
5   <2> add[t1] sK/2 ->6
-   <1> ex-rv2sv sK/1 ->4
3   <$> gvsv(*b) s ->4
4   <$> const(IV 42) s ->5
-   <1> ex-rv2sv sKRM*/1 ->7
6   <$> gvsv(*a) s ->7
```

Each line corresponds to an operator. Null ops appear as *ex-opname*, where *opname* is the op that has been optimized away by perl.

The number on the first row indicates the op's sequence number. It's given in base 36 by default.

The symbol between angle brackets indicates the op's type : for example, <2 is a BINOP, <@ a LISTOP, etc. (see).

The opname may be followed by op-specific information in parentheses (e.g. *gvsv(*b)*), and by targ information in brackets (e.g. *leave[t1]*).

Next come the op flags. The common flags are listed below (). The private flags follow, separated by a slash. For example, *vKP/REFC* means that the *leave* op has public flags *OPf_WANT_VOID*, *OPf_KIDS*, and *OPf_PARENS*, and the private flag *OPpREFCOUNTED*.

Finally an arrow points to the sequence number of the next op.

OPTIONS

Arguments that don't start with a hyphen are taken to be the names of subroutines to print the OPs of; if no such functions are specified, the main body of the program (outside any subroutines, and not including use'd or require'd files) is printed.

-basic

Print OPs in the order they appear in the OP tree (a preorder traversal, starting at the root). The indentation of each OP shows its level in the tree. This mode is the default, so the flag is included simply for completeness.

-exec

Print OPs in the order they would normally execute (for the majority of constructs this is a postorder traversal of the tree, ending at the root). In most cases the OP that usually follows a given OP will appear directly below it; alternate paths are shown by indentation. In cases like loops when control jumps out of a linear path, a 'goto' line is generated.

-tree

Print OPs in a text approximation of a tree, with the root of the tree at the left and 'left-to-right' order of children transformed into 'top-to-bottom'. Because this mode grows both to the right and down, it isn't suitable for large programs (unless you have a very wide terminal).

-compact

Use a tree format in which the minimum amount of space is used for the lines connecting nodes (one character in most cases). This squeezes out a few precious columns of screen real estate.

-loose

Use a tree format that uses longer edges to separate OP nodes. This format tends to look better than the compact one, especially in ASCII, and is the default.

-vt Use tree connecting characters drawn from the VT100 line-drawing set. This looks better if your terminal supports it.

-ascii

Draw the tree with standard ASCII characters like + and |. These don't look as clean as the VT100 characters, but they'll work with almost any terminal (or the horizontal scrolling mode of less(1)) and are suitable for text documentation or email. This is the default.

-main

Include the main program in the output, even if subroutines were also specified.

-basen

Print OP sequence numbers in base *n*. If *n* is greater than 10, the digit for 11 will be 'a', and so on. If *n* is greater than 36, the digit for 37 will be 'A', and so on until 62. Values greater than 62 are not currently supported. The default is 36.

-bigendian

Print sequence numbers with the most significant digit first. This is the usual convention for Arabic numerals, and the default.

-littleendian

Print sequence numbers with the least significant digit first.

-concise

Use the author's favorite set of formatting conventions. This is the default, of course.

-terse

Use formatting conventions that emulate the output of **B::Terse**. The basic mode is almost indistinguishable from the real **B::Terse**, and the exec mode looks very similar, but is in a more logical order and lacks curly brackets. **B::Terse** doesn't have a tree mode, so the tree mode is only vaguely reminiscent of **B::Terse**.

-linenoise

Use formatting conventions in which the name of each OP, rather than being written out in full, is represented by a one- or two-character abbreviation. This is mainly a joke.

-debug

Use formatting conventions reminiscent of **B::Debug**; these aren't very concise at all.

-env

Use formatting conventions read from the environment variables `B_CONCISE_FORMAT`, `B_CONCISE_GOTO_FORMAT`, and `B_CONCISE_TREE_FORMAT`.

FORMATTING SPECIFICATIONS

For each general style ('concise', 'terse', 'linenoise', etc.) there are three specifications: one of how OPs should appear in the basic or exec modes, one of how 'goto' lines should appear (these occur in the exec mode only), and one of how nodes should appear in tree mode. Each has the same format, described below. Any text that doesn't match a special pattern is copied verbatim.

(x(exec_text;basic_text)x)

Generates *exec_text* in exec mode, or *basic_text* in basic mode.

(*(text)*)

Generates one copy of *text* for each indentation level.

(*(text1;text2)*)

Generates one fewer copies of *text1* than the indentation level, followed by one copy of *text2* if the indentation level is more than 0.

(?(text1#varText2)?)

If the value of *var* is true (not empty or zero), generates the value of *var* surrounded by *text1* and *Text2*, otherwise nothing.

#var

Generates the value of the variable *var*.

#varN

Generates the value of *var*, left justified to fill *N* spaces.

~ Any number of tildes and surrounding whitespace will be collapsed to a single space.

The following variables are recognized:

#addr

The address of the OP, in hexadecimal.

#arg

The OP-specific information of the OP (such as the SV for an SVOP, the non-local exit pointers for a LOOP, etc.) enclosed in parentheses.

#class

The B-determined class of the OP, in all caps.

#classsym

A single symbol abbreviating the class of the OP.

#coplabel

The label of the statement or block the OP is the start of, if any.

#exname

The name of the OP, or 'ex-foo' if the OP is a null that used to be a foo.

#extarg

The target of the OP, or nothing for a nulled OP.

#firstaddr

The address of the OP's first child, in hexadecimal.

#flags

The OP's flags, abbreviated as a series of symbols.

#flagval

The numeric value of the OP's flags.

#hyphseq

The sequence number of the OP, or a hyphen if it doesn't have one.

#label

'NEXT', 'LAST', or 'REDO' if the OP is a target of one of those in exec mode, or empty otherwise.

#lastaddr

The address of the OP's last child, in hexadecimal.

#name

The OP's name.

#NAME

The OP's name, in all caps.

#next

The sequence number of the OP's next OP.

#nextaddr

The address of the OP's next OP, in hexadecimal.

#noise

The two-character abbreviation for the OP's name.

#private

The OP's private flags, rendered with abbreviated names if possible.

#privval

The numeric value of the OP's private flags.

#seq

The sequence number of the OP.

#seqnum

The real sequence number of the OP, as a regular number and not adjusted to be relative to the start of the real program. (This will generally be a fairly large number because all of **B::Concise** is compiled before your program is).

#sibaddr

The address of the OP's next youngest sibling, in hexadecimal.

#svaddr

The address of the OP's SV, if it has an SV, in hexadecimal.

#svclass

The class of the OP's SV, if it has one, in all caps (e.g., 'IV').

#svval

The value of the OP's SV, if it has one, in a short human-readable format.

#targ

The numeric value of the OP's targ.

#targarg

The name of the variable the OP's targ refers to, if any, otherwise the letter t followed by the OP's targ in decimal.

#targarglife

Same as **#targarg**, but followed by the COP sequence numbers that delimit the variable's lifetime (or 'end' for a variable in an open scope) for a variable.

#typenum

The numeric value of the OP's type, in decimal.

ABBREVIATIONS**OP flags abbreviations**

v	OPf_WANT_VOID	Want nothing (void context)
s	OPf_WANT_SCALAR	Want single value (scalar context)
l	OPf_WANT_LIST	Want list of any length (list context)
K	OPf_KIDS	There is a firstborn child.
P	OPf_PARENS	This operator was parenthesized. (Or block needs explicit scope entry.)
R	OPf_REF	Certified reference. (Return container, not containee).
M	OPf_MOD	Will modify (lvalue).
S	OPf_STACKED	Some arg is arriving on the stack.
*	OPf_SPECIAL	Do something weird for this op (see op.h)

OP class abbreviations

0	OP (aka BASEOP)	An OP with no children
1	UNOP	An OP with one child
2	BINOP	An OP with two children
	LOGOP	A control branch OP
@	LISTOP	An OP that could have lots of children
/	PMOP	An OP with a regular expression
\$	SVOP	An OP with an SV
"	PVOP	An OP with a string
{	LOOP	An OP that holds pointers for a loop
;	COP	An OP that marks the start of a statement
#	PADOP	An OP with a GV on the pad

Using B::Concise outside of the O framework

It is possible to extend **B::Concise** by using it outside of the **O** framework and providing new styles and new variables.

```
use B::Concise qw(set_style add_callback);
set_style($format, $gotofmt, $treefmt);
add_callback
(
    sub
    {
        my ($h, $op, $level, $format) = @_;
        $h->{variable} = some_func($op);
    }
);
B::Concise::compile(@options)->();
```

You can specify a style by calling the **set_style** subroutine. If you have a new variable in your style, or you want to change the value of an existing variable, you will need to add a callback to specify the value for that

variable.

This is done by calling **add_callback** passing references to any callback subroutines. The subroutines are called in the same order as they are added. Each subroutine is passed four parameters. These are a reference to a hash, the keys of which are the names of the variables and the values of which are their values, the op, the level and the format.

To define your own variables, simply add them to the hash, or change existing values if you need to. The level and format are passed in as references to scalars, but it is unlikely that they will need to be changed or even used.

To see the output, call the subroutine returned by **compile** in the same way that **O** does.

AUTHOR

Stephen McCamant, smcc@CSUA.Berkeley.EDU

NAME

B::Debug – Walk Perl syntax tree, printing debug info about ops

SYNOPSIS

```
perl -MO=Debug[,OPTIONS] foo.pl
```

DESCRIPTION

See *ext/B/README*.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

B::Deparse – Perl compiler backend to produce perl code

SYNOPSIS

```
perl -MO=Deparse[,-uPACKAGE][,-p][,-l]
      [,-sLETTERS][,-xLEVEL] prog.pl
```

DESCRIPTION

B::Deparse is a backend module for the Perl compiler that generates perl source code, based on the internal compiled structure that perl itself creates after parsing a program. The output of B::Deparse won't be exactly the same as the original source, since perl doesn't keep track of comments or whitespace, and there isn't a one-to-one correspondence between perl's syntactical constructions and their compiled form, but it will often be close. When you use the **-p** option, the output also includes parentheses even when they are not required by precedence, which can make it easy to see if perl is parsing your expressions the way you intended.

Please note that this module is mainly new and untested code and is still under development, so it may change in the future.

OPTIONS

As with all compiler backend options, these must follow directly after the '**-MO=Deparse**', separated by a comma but not any white space.

- l** Add '#line' declarations to the output based on the line and file locations of the original code.
- p** Print extra parentheses. Without this option, B::Deparse includes parentheses in its output only when they are needed, based on the structure of your program. With **-p**, it uses parentheses (almost) whenever they would be legal. This can be useful if you are used to LISP, or if you want to see how perl parses your input. If you say

```
if ($var & 0x7f == 65) {print "Gimme an A!"}
print ($which ? $a : $b), "\n";
$name = $ENV{USER} or "Bob";
```

B::Deparse, **-p** will print

```
if (($var & 0)) {
    print('Gimme an A!')
};
(print(($which ? $a : $b)), '???');
(($name = $ENV{'USER'}) or '???')
```

which probably isn't what you intended (the '???' is a sign that perl optimized away a constant value).

- P** Disable prototype checking. With this option, all function calls are deparsed as if no prototype was defined for them. In other words,

```
perl -MO=Deparse,-P -e 'sub foo (\@) { 1 } foo @x'
```

will print

```
sub foo (\@) {
    1;
}
&foo(\@x);
```

making clear how the parameters are actually passed to foo.

- q** Expand double-quoted strings into the corresponding combinations of concatenation, uc, ucfirst, lc, lcfirst, quotemeta, and join. For instance, print

```
print "Hello, $world, @ladies, \u$gentlemen\E, \u\L$me!";
```

as

```
print 'Hello, ' . $world . ', ' . join("$", @ladies) . ', ' .
    . ucfirst($gentlemen) . ', ' . ucfirst(lc $me . '!');
```

Note that the expanded form represents the way perl handles such constructions internally — this option actually turns off the reverse translation that B::Deparse usually does. On the other hand, note that `$x = "$y"` is not the same as `$x = $y`: the former makes the value of `$y` into a string before doing the assignment.

-fFILE

Normally, B::Deparse deparses the main code of a program, and all the subs defined in the same file. To include subs defined in other files, pass the **-f** option with the filename. You can pass the **-f** option several times, to include more than one secondary file. (Most of the time you don't want to use it at all.) You can also use this option to include subs which are defined in the scope of a **#line** directive with two parameters.

-sLETTERS

Tweak the style of B::Deparse's output. The letters should follow directly after the 's', with no space or punctuation. The following options are available:

- C** Cuddle `elsif`, `else`, and `continue` blocks. For example, print

```
if (...) {
    ...
} else {
    ...
}
```

instead of

```
if (...) {
    ...
}
else {
    ...
}
```

The default is not to cuddle.

iNUMBER

Indent lines by multiples of *NUMBER* columns. The default is 4 columns.

- T** Use tabs for each 8 columns of indent. The default is to use only spaces. For instance, if the style options are **-si4T**, a line that's indented 3 times will be preceded by one tab and four spaces; if the options were **-si8T**, the same line would be preceded by three tabs.

vSTRING.

Print *STRING* for the value of a constant that can't be determined because it was optimized away (mnemonic: this happens when a constant is used in void context). The end of the string is marked by a period. The string should be a valid perl expression, generally a constant. Note that unless it's a number, it probably needs to be quoted, and on a command line quotes need to be protected from the shell. Some conventional values include 0, 1, 42, ' ', 'foo', and 'Useless use of constant omitted' (which may need to be **-sv"Useless use of constant omitted".** or something similar depending on your shell). The default is `???`. If you're using B::Deparse on a module or

other file that's require'd, you shouldn't use a value that evaluates to false, since the customary true constant at the end of a module will be in void context when the file is compiled as a main program.

-xLEVEL

Expand conventional syntax constructions into equivalent ones that expose their internal operation. *LEVEL* should be a digit, with higher values meaning more expansion. As with **-q**, this actually involves turning off special cases in B::Deparse's normal operations.

If *LEVEL* is at least 3, for loops will be translated into equivalent while loops with continue blocks; for instance

```
for ($i = 0; $i < 10; ++$i) {
    print $i;
}
```

turns into

```
$i = 0;
while ($i < 10) {
    print $i;
} continue {
    ++$i
}
```

Note that in a few cases this translation can't be perfectly carried back into the source code — if the loop's initializer declares a my variable, for instance, it won't have the correct scope outside of the loop.

If *LEVEL* is at least 7, if statements will be translated into equivalent expressions using &&, ?: and do {}; for instance

```
print 'hi' if $nice;
if ($nice) {
    print 'hi';
}
if ($nice) {
    print 'hi';
} else {
    print 'bye';
}
```

turns into

```
$nice and print 'hi';
$nice and do { print 'hi' };
$nice ? do { print 'hi' } : do { print 'bye' };
```

Long sequences of elsifs will turn into nested ternary operators, which B::Deparse doesn't know how to indent nicely.

USING B::Deparse AS A MODULE

Synopsis

```
use B::Deparse;
$dparse = B::Deparse->new("-p", "-sC");
$body = $dparse->coderef2text(\&func);
eval "sub func $body"; # the inverse operation
```

Description

B::Deparse can also be used on a sub-by-sub basis from other perl programs.

new

```
$deparse = B::Deparse->new(OPTIONS)
```

Create an object to store the state of a deparsing operation and any options. The options are the same as those that can be given on the command line (see [/OPTIONS](#)); options that are separated by commas after **-MO=Deparse** should be given as separate strings. Some options, like **-u**, don't make sense for a single subroutine, so don't pass them.

ambient_pragmas

```
$deparse->ambient_pragmas(strict => 'all', '$[' => $[);
```

The compilation of a subroutine can be affected by a few compiler directives, **pragmas**. These are:

- use strict;
- use warnings;
- Assigning to the special variable \$[
- use integer;
- use bytes;
- use utf8;
- use re;

Ordinarily, if you use B::Deparse on a subroutine which has been compiled in the presence of one or more of these pragmas, the output will include statements to turn on the appropriate directives. So if you then compile the code returned by `coderef2text`, it will behave the same way as the subroutine which you deparsed.

However, you may know that you intend to use the results in a particular context, where some pragmas are already in scope. In this case, you use the **ambient_pragmas** method to describe the assumptions you wish to make.

Not all of the options currently have any useful effect. See [/BUGS](#) for more details.

The parameters it accepts are:

strict

Takes a string, possibly containing several values separated by whitespace. The special values "all" and "none" mean what you'd expect.

```
$deparse->ambient_pragmas(strict => 'subs refs');
```

`$[` Takes a number, the value of the array base `$[`.

bytes

utf8

integer

If the value is true, then the appropriate pragma is assumed to be in the ambient scope, otherwise not.

re Takes a string, possibly containing a whitespace-separated list of values. The values "all" and "none" are special. It's also permissible to pass an array reference here.

```
$deparse->ambient_pragmas(re => 'eval');
```

warnings

Takes a string, possibly containing a whitespace-separated list of values. The values "all" and "none" are special, again. It's also permissible to pass an array reference here.

```
$deparser->ambient_pragmas(warnings => [qw[void io]]);
```

If one of the values is the string "FATAL", then all the warnings in that list will be considered fatal, just as with the **warnings** pragma itself. Should you need to specify that some warnings are fatal, and others are merely enabled, you can pass the **warnings** parameter twice:

```
$deparser->ambient_pragmas(
    warnings => 'all',
    warnings => [FATAL => qw[void io/]],
);
```

See [perllexwarn](#) for more information about lexical warnings.

hint_bits

warning_bits

These two parameters are used to specify the ambient pragmas in the format used by the special variables `^H` and `^WARNING_BITS`.

They exist principally so that you can write code like:

```
{ my ($hint_bits, $warning_bits);
BEGIN { ($hint_bits, $warning_bits) = (^H, ^WARNING_BITS) }
$deparser->ambient_pragmas (
    hint_bits    => $hint_bits,
    warning_bits => $warning_bits,
    '$['        => 0 + $[
); }
```

which specifies that the ambient pragmas are exactly those which are in scope at the point of calling.

coderef2text

```
$body = $deparse->coderef2text(\&func)
$body = $deparse->coderef2text(sub ($$) { ... })
```

Return source code for the body of a subroutine (a block, optionally preceded by a prototype in parens), given a reference to the sub. Because a subroutine can have no names, or more than one name, this method doesn't return a complete subroutine definition — if you want to eval the result, you should prepend "sub subname ", or "sub " for an anonymous function constructor. Unless the sub was defined in the main:: package, the code will include a package declaration.

BUGS

- The only pragmas to be completely supported are: `use warnings`, `use strict 'refs'`, `use bytes`, and `use integer`. (`$[`, which behaves like a pragma, is also supported.)

Excepting those listed above, we're currently unable to guarantee that B::Deparse will produce a pragma at the correct point in the program. Since the effects of pragmas are often lexically scoped, this can mean that the pragma holds sway over a different portion of the program than in the input file.

- In fact, the above is a specific instance of a more general problem: we can't guarantee to produce BEGIN blocks or use declarations in exactly the right place. So if you use a module which affects compilation (such as by over-riding keywords, overloading constants or whatever) then the output code might not work as intended.

This is the most serious outstanding problem, and will be very hard to fix.

- If a keyword is over-ridden, and your program explicitly calls the built-in version by using `CORE::keyword`, the output of B::Deparse will not reflect this. If you run the resulting code, it will call the over-ridden version rather than the built-in one. (Maybe there should be an option to **always** print keyword calls as `CORE:::name`.)

- `sort foo (1, 2, 3)` comes out as `sort (foo 1, 2, 3)`, which causes perl to issue a warning.

The obvious fix doesn't work, because these are different:

```
print (FOO 1, 2, 3), 4, 5, 6;  
print FOO (1, 2, 3), 4, 5, 6;
```

- Constants (other than simple strings or numbers) don't work properly. Pathological examples that fail (and probably always will) include:

```
use constant E2BIG => ($!=7);  
use constant x=>\$x; print x
```

The following could (and should) be made to work:

```
use constant regex => qr/blah/;  
print regex;
```

- An input file that uses source filtering probably won't be deparsed into runnable code, because it will still include the **use** declaration for the source filtering module, even though the code that is produced is already ordinary Perl which shouldn't be filtered again.
- There are probably many more bugs on non-ASCII platforms (EBCDIC).

AUTHOR

Stephen McCamant <smcc@CSUA.Berkeley.EDU>, based on an earlier version by Malcolm Beattie <mbeattie@sable.ox.ac.uk>, with contributions from Gisle Aas, James Duncan, Albert Dvornik, Robin Houston, Hugo van der Sanden, Gurusamy Sarathy, Nick Ing-Simmons, and Rafael Garcia-Suarez.

NAME

B::Disassembler – Disassemble Perl bytecode

SYNOPSIS

```
use Disassembler;
```

DESCRIPTION

See *ext/B/B/Disassembler.pm*.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

B::Lint – Perl lint

SYNOPSIS

```
perl -MO=Lint[,OPTIONS] foo.pl
```

DESCRIPTION

The B::Lint module is equivalent to an extended version of the **-w** option of **perl**. It is named after the program **lint** which carries out a similar process for C programs.

OPTIONS AND LINT CHECKS

Option words are separated by commas (not whitespace) and follow the usual conventions of compiler backend options. Following any options (indicated by a leading **-**) come lint check arguments. Each such argument (apart from the special **all** and **none** options) is a word representing one possible lint check (turning on that check) or is **no-foo** (turning off that check). Before processing the check arguments, a standard list of checks is turned on. Later options override earlier ones. Available options are:

context Produces a warning whenever an array is used in an implicit scalar context. For example, both of the lines

```
$foo = length(@bar);  
$foo = @bar;
```

will elicit a warning. Using an explicit **scalar()** silences the warning. For example,

```
$foo = scalar(@bar);
```

implicit-read and implicit-write

These options produce a warning whenever an operation implicitly reads or (respectively) writes to one of Perl's special variables. For example, **implicit-read** will warn about these:

```
/foo/;
```

and **implicit-write** will warn about these:

```
s/foo/bar/;
```

Both **implicit-read** and **implicit-write** warn about this:

```
for (@a) { ... }
```

dollar-underscore

This option warns whenever **\$_** is used either explicitly anywhere or as the implicit argument of a **print** statement.

private-names

This option warns on each use of any variable, subroutine or method name that lives in a non-current package but begins with an underscore ("**_**"). Warnings aren't issued for the special case of the single character name "**_**" by itself (e.g. **\$_** and **@_**).

undefined Subs

This option warns whenever an undefined subroutine is invoked. This option will only catch explicitly invoked subroutines such as **foo()** and not indirect invocations such as **&\$subref()** or **\$obj->meth()**. Note that some programs or modules delay definition of subs until runtime by means of the **AUTOLOAD** mechanism.

regex-variables

This option warns whenever one of the regex variables **\$'**, **\$&** or **\$'** is used. Any occurrence of any of these variables in your program can slow your whole program down. See [perlre](#) for details.

all Turn all warnings on.

none Turn all warnings off.

NON LINT-CHECK OPTIONS

-u Package

Normally, Lint only checks the main code of the program together with all subs defined in package main. The **-u** option lets you include other package names whose subs are then checked by Lint.

BUGS

This is only a very preliminary version.

This module doesn't work correctly on thread-enabled perls.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk.

NAME

B::Showlex – Show lexical variables used in functions or files

SYNOPSIS

```
perl -MO=Showlex[,SUBROUTINE] foo.pl
```

DESCRIPTION

When a subroutine name is provided in `OPTIONS`, prints the lexical variables used in that subroutine. Otherwise, prints the file-scope lexicals in the file.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

B::Stackobj – Helper module for CC backend

SYNOPSIS

```
use B::Stackobj;
```

DESCRIPTION

See *ext/B/README*.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

B::Stash – show what stashes are loaded

NAME

B::Terse – Walk Perl syntax tree, printing terse info about ops

SYNOPSIS

```
perl -MO=Terse[,OPTIONS] foo.pl
```

DESCRIPTION

See *ext/B/README*.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

B::Xref – Generates cross reference reports for Perl programs

SYNOPSIS

```
perl -MO=Xref[,OPTIONS] foo.pl
```

DESCRIPTION

The B::Xref module is used to generate a cross reference listing of all definitions and uses of variables, subroutines and formats in a Perl program. It is implemented as a backend for the Perl compiler.

The report generated is in the following format:

```
File filename1
  Subroutine subname1
    Package package1
      object1      line numbers
      object2      line numbers
      ...
    Package package2
    ...
```

Each **File** section reports on a single file. Each **Subroutine** section reports on a single subroutine apart from the special cases "(definitions)" and "(main)". These report, respectively, on subroutine definitions found by the initial symbol table walk and on the main part of the program or module external to all subroutines.

The report is then grouped by the **Package** of each variable, subroutine or format with the special case "(lexicals)" meaning lexical variables. Each **object** name (implicitly qualified by its containing **Package**) includes its type character(s) at the beginning where possible. Lexical variables are easier to track and even included dereferencing information where possible.

The `line numbers` are a comma separated list of line numbers (some preceded by code letters) where that object is used in some way. Simple uses aren't preceded by a code letter. Introductions (such as where a lexical is first defined with `my`) are indicated with the letter "i". Subroutine and method calls are indicated by the character "&". Subroutine definitions are indicated by "s" and format definitions by "f".

OPTIONS

Option words are separated by commas (not whitespace) and follow the usual conventions of compiler backend options.

`-oFILENAME`

Directs output to `FILENAME` instead of standard output.

`-r` Raw output. Instead of producing a human-readable report, outputs a line in machine-readable form for each definition/use of a variable/sub/format.

`-d` Don't output the "(definitions)" sections.

`-D[tO]` (Internal) debug options, probably only useful if `-r` included. The `t` option prints the object on the top of the stack as it's being tracked. The `O` option prints each operator as it's being processed in the execution order of the program.

BUGS

Non-lexical variables are quite difficult to track through a program. Sometimes the type of a non-lexical variable's use is impossible to determine. Introductions of non-lexical non-scalars don't seem to be reported properly.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk.

NAME

O – Generic interface to Perl Compiler backends

SYNOPSIS

```
perl -MO=[-q,]Backend[,OPTIONS] foo.pl
```

DESCRIPTION

This is the module that is used as a frontend to the Perl Compiler.

If you pass the `-q` option to the module, then the `STDOUT` filehandle will be redirected into the variable `$O::BEGIN_output` during compilation. This has the effect that any output printed to `STDOUT` by `BEGIN` blocks or `use'd` modules will be stored in this variable rather than printed. It's useful with those backends which produce output themselves (`Deparse`, `Concise` etc), so that their output is not confused with that generated by the code being compiled.

The `-qq` option behaves like `-q`, except that it also closes `STDERR` after deparsing has finished. This suppresses the "Syntax OK" message normally produced by `perl`.

CONVENTIONS

Most compiler backends use the following conventions: `OPTIONS` consists of a comma-separated list of words (no white-space). The `-v` option usually puts the backend into verbose mode. The `-ofile` option generates output to `file` instead of `stdout`. The `-D` option followed by various letters turns on various internal debugging flags. See the documentation for the desired backend (named `B::Backend` for the example above) to find out about that backend.

IMPLEMENTATION

This section is only necessary for those who want to write a compiler backend module that can be used via this module.

The command-line mentioned in the **SYNOPSIS** section corresponds to the Perl code

```
use O ("Backend", OPTIONS);
```

The `import` function which that calls loads in the appropriate `B::Backend` module and calls the `compile` function in that package, passing it `OPTIONS`. That function is expected to return a sub reference which we'll call `CALLBACK`. Next, the "compile-only" flag is switched on (equivalent to the command-line option `-c`) and a `CHECK` block is registered which calls `CALLBACK`. Thus the main Perl program mentioned on the command-line is read in, parsed and compiled into internal syntax tree form. Since the `-c` flag is set, the program does not start running (excepting `BEGIN` blocks of course) but the `CALLBACK` function registered by the compiler backend is called.

In summary, a compiler backend module should be called `"B::Foo"` for some `foo` and live in the appropriate directory for that name. It should define a function called `compile`. When the user types

```
perl -MO=Foo,OPTIONS foo.pl
```

that function is called and is passed those `OPTIONS` (split on commas). It should return a sub ref to the main compilation function. After the user's program is loaded and parsed, that returned sub ref is invoked which can then go ahead and do the compilation, usually by making use of the `B` module's functionality.

BUGS

The `-q` and `-qq` options don't work correctly if `perl` isn't compiled with `PerlIO` support : `STDOUT` will be closed instead of being redirected to `$O::BEGIN_output`.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk

NAME

base – Establish IS–A relationship with base class at compile time

SYNOPSIS

```
package Baz;  
use base qw(Foo Bar);
```

DESCRIPTION

Roughly similar in effect to

```
BEGIN {  
    require Foo;  
    require Bar;  
    push @ISA, qw(Foo Bar);  
}
```

Will also initialize the %FIELDS hash if one of the base classes has it. Multiple inheritance of %FIELDS is not supported. The ‘base’ pragma will croak if multiple base classes have a %FIELDS hash. See [fields](#) for a description of this feature.

When strict ‘vars’ is in scope *base* also let you assign to @ISA without having to declare @ISA with the ‘vars’ pragma first.

If any of the base classes are not loaded yet, *base* silently requires them. Whether to require a base class package is determined by the absence of a global \$VERSION in the base package. If \$VERSION is not detected even after loading it, <base will define \$VERSION in the base package, setting it to the string -1, set by base.pm.

HISTORY

This module was introduced with Perl 5.004_04.

SEE ALSO

[fields](#)

NAME

Benchmark – benchmark running times of Perl code

SYNOPSIS

```
use Benchmark qw(:all) ;

timethis ($count, "code");

# Use Perl code in strings...
timethese($count, {
    'Name1' => '...code1...',
    'Name2' => '...code2...',
});

# ... or use subroutine references.
timethese($count, {
    'Name1' => sub { ...code1... },
    'Name2' => sub { ...code2... },
});

# cmpthese can be used both ways as well
cmpthese($count, {
    'Name1' => '...code1...',
    'Name2' => '...code2...',
});

cmpthese($count, {
    'Name1' => sub { ...code1... },
    'Name2' => sub { ...code2... },
});

# ...or in two stages
$results = timethese($count,
    {
        'Name1' => sub { ...code1... },
        'Name2' => sub { ...code2... },
    },
    'none'
);
cmpthese( $results );

$t = timeit($count, '...other code...')
print "$count loops of other code took:",timestr($t),"\n";

$t = countit($time, '...other code...')
$count = $t->iters ;
print "$count loops of other code took:",timestr($t),"\n";
```

DESCRIPTION

The Benchmark module encapsulates a number of routines to help you figure out how long it takes to execute some code.

timethis – run a chunk of code several times

timethese – run several chunks of code several times

cmpthese – print results of timethese as a comparison chart

timeit – run a chunk of code and see how long it goes

countit – see how many times a chunk of code runs in a given time

Methods

- new** Returns the current time. Example:
- ```
use Benchmark;
$t0 = new Benchmark;
... your code here ...
$t1 = new Benchmark;
$td = timediff($t1, $t0);
print "the code took:", timestr($td), "\n";
```
- debug** Enables or disable debugging by setting the `$Benchmark::Debug` flag:
- ```
debug Benchmark 1;
$t = timeit(10, ' 5 ** $Global ');
debug Benchmark 0;
```
- iters** Returns the number of iterations.

Standard Exports

The following routines will be exported into your namespace if you use the Benchmark module:

timeit(COUNT, CODE)

Arguments: COUNT is the number of times to run the loop, and CODE is the code to run. CODE may be either a code reference or a string to be eval'd; either way it will be run in the caller's package.

Returns: a Benchmark object.

timethis (COUNT, CODE, [TITLE, [STYLE]])

Time COUNT iterations of CODE. CODE may be a string to eval or a code reference; either way the CODE will run in the caller's package. Results will be printed to STDOUT as TITLE followed by the times. TITLE defaults to "timethis COUNT" if none is provided. STYLE determines the format of the output, as described for `timestr()` below.

The COUNT can be zero or negative: this means the *minimum number of CPU seconds* to run. A zero signifies the default of 3 seconds. For example to run at least for 10 seconds:

```
timethis(-10, $code)
```

or to run two pieces of code tests for at least 3 seconds:

```
timethese(0, { test1 => '...', test2 => '...' })
```

CPU seconds is, in UNIX terms, the user time plus the system time of the process itself, as opposed to the real (wallclock) time and the time spent by the child processes. Less than 0.1 seconds is not accepted (-0.01 as the count, for example, will cause a fatal runtime exception).

Note that the CPU seconds is the **minimum** time: CPU scheduling and other operating system factors may complicate the attempt so that a little bit more time is spent. The benchmark output will, however, also tell the number of \$code runs/second, which should be a more interesting number than the actually spent seconds.

Returns a Benchmark object.

timethese (COUNT, CODEHASHREF, [STYLE])

The CODEHASHREF is a reference to a hash containing names as keys and either a string to eval or a code reference for each value. For each (KEY, VALUE) pair in the CODEHASHREF, this routine will call

```
timethis(COUNT, VALUE, KEY, STYLE)
```

The routines are called in string comparison order of KEY.

The COUNT can be zero or negative, see `timethis()`.

Returns a hash of Benchmark objects, keyed by name.

```
timediff ( T1, T2 )
```

Returns the difference between two Benchmark times as a Benchmark object suitable for passing to `timestr()`.

```
timestr ( TIMEDIFF, [ STYLE, [ FORMAT ] ] )
```

Returns a string that formats the times in the TIMEDIFF object in the requested STYLE. TIMEDIFF is expected to be a Benchmark object similar to that returned by `timediff()`.

STYLE can be any of 'all', 'none', 'noc', 'nop' or 'auto'. 'all' shows each of the 5 times available ('wallclock' time, user time, system time, user time of children, and system time of children). 'noc' shows all except the two children times. 'nop' shows only wallclock and the two children times. 'auto' (the default) will act as 'all' unless the children times are both zero, in which case it acts as 'noc'. 'none' prevents output.

FORMAT is the [printf\(3\)](#)-style format specifier (without the leading '%') to use to print the times. It defaults to '5.2f'.

Optional Exports

The following routines will be exported into your namespace if you specifically ask that they be imported:

```
clearcache ( COUNT )
```

Clear the cached time for COUNT rounds of the null loop.

```
clearallcache ( )
```

Clear all cached times.

```
cmpthese ( COUNT, CODEHASHREF, [ STYLE ] )
```

```
cmpthese ( RESULTSHASHREF, [ STYLE ] )
```

Optionally calls `timethese()`, then outputs comparison chart. This:

```
cmpthese( -1, { a => "++\${i}", b => "\${i} *= 2" } ) ;
```

outputs a chart like:

```

      Rate      b      a
b 2831802/s  -- -61%
a 7208959/s 155%  --
```

This chart is sorted from slowest to fastest, and shows the percent speed difference between each pair of tests.

`c<cmpthese` can also be passed the data structure that `timethese()` returns:

```

$results = timethese( -1, { a => "++\${i}", b => "\${i} *= 2" } ) ;
cmpthese( $results );
```

in case you want to see both sets of results.

Returns a reference to an ARRAY of rows, each row is an ARRAY of cells from the above chart, including labels. This:

```
my $rows = cmpthese( -1, { a => '++\${i}', b => '\${i} *= 2' }, "none" );
```

returns a data structure like:

```
[
```

```

[ ' ', 'Rate', 'b', 'a' ],
[ 'b', '2885232/s', '--', '-59%' ],
[ 'a', '7099126/s', '146%', '--' ],
]

```

NOTE: This result value differs from previous versions, which returned the `timethese()` result structure. If you want that, just use the two statement `timethese...cmpthese` idiom shown above.

Incidentally, note the variance in the result values between the two examples; this is typical of benchmarking. If this were a real benchmark, you would probably want to run a lot more iterations.

`countit(TIME, CODE)`

Arguments: `TIME` is the minimum length of time to run `CODE` for, and `CODE` is the code to run. `CODE` may be either a code reference or a string to be eval'd; either way it will be run in the caller's package.

`TIME` is *not* negative. `countit()` will run the loop many times to calculate the speed of `CODE` before running it for `TIME`. The actual time run for will usually be greater than `TIME` due to system clock resolution, so it's best to look at the number of iterations divided by the times that you are concerned with, not just the iterations.

Returns: a Benchmark object.

`disablecache()`

Disable caching of timings for the null loop. This will force Benchmark to recalculate these timings for each new piece of code timed.

`enablecache()`

Enable caching of timings for the null loop. The time taken for `COUNT` rounds of the null loop will be calculated only once for each different `COUNT` used.

`timesum(T1, T2)`

Returns the sum of two Benchmark times as a Benchmark object suitable for passing to `timestr()`.

NOTES

The data is stored as a list of values from the time and times functions:

```
($real, $user, $system, $children_user, $children_system, $iters)
```

in seconds for the whole loop (not divided by the number of rounds).

The timing is done using `time(3)` and `times(3)`.

Code is executed in the caller's package.

The time of the null loop (a loop with the same number of rounds but empty loop body) is subtracted from the time of the real loop.

The null loop times can be cached, the key being the number of rounds. The caching can be controlled using calls like these:

```

clearcache($key);
clearallcache();

disablecache();
enablecache();

```

Caching is off by default, as it can (usually slightly) decrease accuracy and does not usually noticeably affect runtimes.

EXAMPLES

For example,

```

use Benchmark qw( cmpthese ) ;
$x = 3;
cmpthese( -5, {
    a => sub{ $x*$x },
    b => sub{ $x**2 },
} );

```

outputs something like this:

```

Benchmark: running a, b, each for at least 5 CPU seconds...
      Rate      b      a
b 1559428/s  -- -62%
a 4152037/s 166%  --

```

while

```

use Benchmark qw( timethese cmpthese ) ;
$x = 3;
$r = timethese( -5, {
    a => sub{ $x*$x },
    b => sub{ $x**2 },
} );
cmpthese $r;

```

outputs something like this:

```

Benchmark: running a, b, each for at least 5 CPU seconds...
      a: 10 wallclock secs ( 5.14 usr +  0.13 sys =  5.27 CPU) @ 3835055.60/s
      b:  5 wallclock secs ( 5.41 usr +  0.00 sys =  5.41 CPU) @ 1574944.92/s
      Rate      b      a
b 1574945/s  -- -59%
a 3835056/s 144%  --

```

INHERITANCE

Benchmark inherits from no other class, except of course for Exporter.

CAVEATS

Comparing eval'd strings with code references will give you inaccurate results: a code reference will show a slightly slower execution time than the equivalent eval'd string.

The real time timing is done using time(2) and the granularity is therefore only one second.

Short tests may produce negative figures because perl can appear to take longer to execute the empty loop than a short test; try:

```
timethis(100, '1');
```

The system time of the null loop might be slightly more than the system time of the loop with the actual code and therefore the difference might end up being < 0.

SEE ALSO

[Devel::DProf](#) – a Perl code profiler

AUTHORS

Jarkko Hietaniemi <jhi@iki.fi>, Tim Bunce <Tim.Bunce@ig.co.uk>

MODIFICATION HISTORY

September 8th, 1994; by Tim Bunce.

March 28th, 1997; by Hugo van der Sanden: added support for code references and the already documented 'debug' method; revamped documentation.

April 04–07th, 1997: by Jarkko Hietaniemi, added the run–for–some–time functionality.

September, 1999; by Barrie Slaymaker: math fixes and accuracy and efficiency tweaks. Added `cmpthese()`. A result is now returned from `timethese()`. Exposed `countit()` (was `runfor()`).

December, 2001; by Nicholas Clark: make `timestr()` recognise the style 'none' and return an empty string. If `cmpthese` is calling `timethese`, make it pass the style in. (so that 'none' will suppress output). Make sub `new` dump its debugging output to `STDERR`, to be consistent with everything else. All bugs found while writing a regression test.

NAME

bigint – Transparent big integer support for Perl

SYNOPSIS

```
use bigint;

$x = 2 + 4.5, "\n";           # BigInt 6
print 2 ** 512;               # really is what you think it is
```

DESCRIPTION

All operators (including basic math operations) are overloaded. Integer constants are created as proper BigInts.

Floating point constants are truncated to integer. All results are also truncated.

OPTIONS

bigint recognizes some options that can be passed while loading it via use. The options can (currently) be either a single letter form, or the long form. The following options exist:

a or accuracy

This sets the accuracy for all math operations. The argument must be greater than or equal to zero. See Math::BigInt's `bround()` function for details.

```
perl -Mbigint=a,2 -le 'print 12345+1'
```

p or precision

This sets the precision for all math operations. The argument can be any integer. Negative values mean a fixed number of digits after the dot, and are <Bignored> since all operations happen in integer space. A positive value rounds to this digit left from the dot. 0 or 1 mean round to integer and are ignore like negative values.

See Math::BigInt's `bfround()` function for details.

```
perl -Mbignum=p,5 -le 'print 123456789+123'
```

t or trace

This enables a trace mode and is primarily for debugging bigint or Math::BigInt.

l or lib

Load a different math lib, see [MATH LIBRARY](#).

```
perl -Mbigint=l,GMP -e 'print 2 ** 512'
```

Currently there is no way to specify more than one library on the command line. This will be hopefully fixed soon ;)

v or version

This prints out the name and version of all modules used and then exits.

```
perl -Mbigint=v -e ''
```

MATH LIBRARY

Math with the numbers is done (by default) by a module called Math::BigInt::Calc. This is equivalent to saying:

```
use bigint lib => 'Calc';
```

You can change this by using:

```
use bigint lib => 'BitVect';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and when this also fails, revert to Math::BigInt::Calc:


```
use bigint lib => 'Foo,Math::BigInt::Bar';
```

Please see respective module documentation for further details.

INTERNAL FORMAT

The numbers are stored as objects, and their internals might change at anytime, especially between math operations. The objects also might belong to different classes, like `Math::BigInt`, or `Math::BigInt::Lite`. Mixing them together, even with normal scalars is not extraordinary, but normal and expected.

You should not depend on the internal format, all accesses must go through accessor methods. E.g. looking at `$x->{sign}` is not a bright idea since there is no guaranty that the object in question has such a hash key, nor is a hash underneath at all.

SIGN

The sign is either `+`, `-`, `NaN`, `+inf` or `-inf` and stored separately. You can access it with the `sign()` method.

A sign of `NaN` is used to represent the result when input arguments are not numbers or as a result of `0/0`. `+inf` and `-inf` represent plus respectively minus infinity. You will get `+inf` when dividing a positive number by 0, and `-inf` when dividing any negative number by 0.

METHODS

Since all numbers are now objects, you can use all functions that are part of the `BigInt` API. You can only use the `bxxx()` notation, and not the `fxxx()` notation, though.

MODULES USED

`bigint` is just a thin wrapper around various modules of the `Math::BigInt` family. Think of it as the head of the family, who runs the shop, and orders the others to do the work.

The following modules are currently used by `bigint`:

```
Math::BigInt::Lite      (for speed, and only if it is loadable)
Math::BigInt
```

EXAMPLES

Some cool command line examples to impress the Python crowd ;) You might want to compare them to the results under `-Mbignum` or `-Mbigrat`:

```
perl -Mbigint -le 'print sqrt(33)'
perl -Mbigint -le 'print 2*255'
perl -Mbigint -le 'print 4.5+2*255'
perl -Mbigint -le 'print 3/7 + 5/7 + 8/3'
perl -Mbigint -le 'print 123->is_odd()'
perl -Mbigint -le 'print log(2)'
perl -Mbigint -le 'print 2 ** 0.5'
perl -Mbigint=a,65 -le 'print 2 ** 0.2'
```

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Especially *bigrat* as in `perl -Mbigrat -le 'print 1/3+1/4'` and *bignum* as in `perl -Mbignum -le 'print sqrt(2)'`.

Math::BigInt, *Math::BigRat* and *Math::Big* as well as *Math::BigInt::BitVect*, *Math::BigInt::Pari* and *Math::BigInt::GMP*.

AUTHORS

(C) by Tels <http://bloodgate.com/> in early 2002.

NAME

bignum – Transparent BigNumber support for Perl

SYNOPSIS

```
use bignum;

$x = 2 + 4.5, "\n";           # BigFloat 6.5
print 2 ** 512 * 0.1;         # really is what you think it is
```

DESCRIPTION

All operators (including basic math operations) are overloaded. Integer and floating-point constants are created as proper BigInts or BigFloats, respectively.

OPTIONS

bignum recognizes some options that can be passed while loading it via use. The options can (currently) be either a single letter form, or the long form. The following options exist:

a or accuracy

This sets the accuracy for all math operations. The argument must be greater than or equal to zero. See Math::BigInt's `bround()` function for details.

```
perl -Mbignum=a,50 -le 'print sqrt(20)'
```

p or precision

This sets the precision for all math operations. The argument can be any integer. Negative values mean a fixed number of digits after the dot, while a positive value rounds to this digit left from the dot. 0 or 1 mean round to integer. See Math::BigInt's `bfround()` function for details.

```
perl -Mbignum=p,-50 -le 'print sqrt(20)'
```

t or trace

This enables a trace mode and is primarily for debugging bignum or Math::BigInt/Math::BigFloat.

l or lib

Load a different math lib, see [MATH LIBRARY](#).

```
perl -Mbignum=l,GMP -e 'print 2 ** 512'
```

Currently there is no way to specify more than one library on the command line. This will be hopefully fixed soon ;)

v or version

This prints out the name and version of all modules used and then exits.

```
perl -Mbignum=v -e ''
```

MATH LIBRARY

Math with the numbers is done (by default) by a module called Math::BigInt::Calc. This is equivalent to saying:

```
use bignum lib => 'Calc';
```

You can change this by using:

```
use bignum lib => 'BitVect';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and when this also fails, revert to Math::BigInt::Calc:

```
use bignum lib => 'Foo,Math::BigInt::Bar';
```

Please see respective module documentation for further details.

INTERNAL FORMAT

The numbers are stored as objects, and their internals might change at anytime, especially between math operations. The objects also might belong to different classes, like `Math::BigInt`, or `Math::BigFloat`. Mixing them together, even with normal scalars is not extraordinary, but normal and expected.

You should not depend on the internal format, all accesses must go through accessor methods. E.g. looking at `$x->{sign}` is not a bright idea since there is no guaranty that the object in question has such a hashkey, nor is a hash underneath at all.

SIGN

The sign is either `‘+’`, `‘-’`, `‘NaN’`, `‘+inf’` or `‘-inf’` and stored separately. You can access it with the `sign()` method.

A sign of `‘NaN’` is used to represent the result when input arguments are not numbers or as a result of `0/0`. `‘+inf’` and `‘-inf’` represent plus respectively minus infinity. You will get `‘+inf’` when dividing a positive number by 0, and `‘-inf’` when dividing any negative number by 0.

METHODS

Since all numbers are now objects, you can use all functions that are part of the `BigInt` or `BigFloat` API. It is wise to use only the `bxxx()` notation, and not the `fxxx()` notation, though. This makes it possible that the underlying object might morph into a different class than `BigFloat`.

MODULES USED

`bignum` is just a thin wrapper around various modules of the `Math::BigInt` family. Think of it as the head of the family, who runs the shop, and orders the others to do the work.

The following modules are currently used by `bignum`:

```
Math::BigInt::Lite      (for speed, and only if it is loadable)
Math::BigInt
Math::BigFloat
```

EXAMPLES

Some cool command line examples to impress the Python crowd ;)

```
perl -Mbignum -le 'print sqrt(33)'
perl -Mbignum -le 'print 2*255'
perl -Mbignum -le 'print 4.5+2*255'
perl -Mbignum -le 'print 3/7 + 5/7 + 8/3'
perl -Mbignum -le 'print 123->is_odd()'
perl -Mbignum -le 'print log(2)'
perl -Mbignum -le 'print 2 ** 0.5'
perl -Mbignum=a,65 -le 'print 2 ** 0.2'
```

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Especially *bigrat* as in `perl -Mbigrat -le 'print 1/3+1/4'`.

Math::BigFloat, *Math::BigInt*, *Math::BigRat* and *Math::Big* as well as *Math::BigInt::BitVect*, *Math::BigInt::Pari* and *Math::BigInt::GMP*.

AUTHORS

(C) by Tels <http://bloodgate.com/> in early 2002.

NAME

bigrat – Transparent BigNumber/BigRational support for Perl

SYNOPSIS

```
use bigrat;

$x = 2 + 4.5, "\n";           # BigFloat 6.5
print 1/3 + 1/4, "\n";       # produces 7/12
```

DESCRIPTION

All operators (including basic math operations) are overloaded. Integer and floating-point constants are created as proper BigInts or BigFloats, respectively.

Other than *bignum*, this module upgrades to Math::BigRat, meaning that instead of 2.5 you will get 2+1/2 as output.

MODULES USED

bigrat is just a thin wrapper around various modules of the Math::BigInt family. Think of it as the head of the family, who runs the shop, and orders the others to do the work.

The following modules are currently used by bignum:

```
Math::BigInt::Lite      (for speed, and only if it is loadable)
Math::BigInt
Math::BigFloat
Math::BigRat
```

MATH LIBRARY

Math with the numbers is done (by default) by a module called Math::BigInt::Calc. This is equivalent to saying:

```
use bigrat lib => 'Calc';
```

You can change this by using:

```
use bigrat lib => 'BitVect';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and when this also fails, revert to Math::BigInt::Calc:

```
use bigrat lib => 'Foo,Math::BigInt::Bar';
```

Please see respective module documentation for further details.

SIGN

The sign is either '+', '-', 'NaN', '+inf' or '-inf' and stored separately.

A sign of 'NaN' is used to represent the result when input arguments are not numbers or as a result of 0/0. '+inf' and '-inf' represent plus respectively minus infinity. You will get '+inf' when dividing a positive number by 0, and '-inf' when dividing any negative number by 0.

METHODS

Since all numbers are not objects, you can use all functions that are part of the BigInt or BigFloat API. It is wise to use only the `bxxx()` notation, and not the `fxxx()` notation, though. This makes you independent on the fact that the underlying object might morph into a different class than BigFloat.

EXAMPLES

```
perl -Mbigrat -le 'print sqrt(33)'
perl -Mbigrat -le 'print 2*255'
perl -Mbigrat -le 'print 4.5+2*255'
perl -Mbigrat -le 'print 3/7 + 5/7 + 8/3'
```

```
perl -Mbigrat -le 'print 12->is_odd()';
```

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Especially *bignum*.

Math::BigFloat, *Math::BigInt*, *Math::BigRat* and *Math::Big* as well as *Math::BigInt::BitVect*, *Math::BigInt::Pari* and *Math::BigInt::GMP*.

AUTHORS

(C) by Tels <http://bloodgate.com/> in early 2002.

NAME

blib – Use MakeMaker's uninstalled version of a package

SYNOPSIS

```
perl -Mblib script [args...]  
perl -Mblib=dir script [args...]
```

DESCRIPTION

Looks for MakeMaker-like *'blib'* directory structure starting in *dir* (or current directory) and working back up to five levels of *'..'*.

Intended for use on command line with **-M** option as a way of testing arbitrary scripts against an uninstalled version of a package.

However it is possible to :

```
use blib;  
or  
use blib '..';
```

etc. if you really must.

BUGS

Pollutes global name space for development only task.

AUTHOR

Nick Ing-Simmons nik@tiuk.ti.com

NAME

ByteLoader – load byte compiled perl code

SYNOPSIS

```
use ByteLoader 0.04;  
<byte code>  
  
use ByteLoader 0.04;  
<byte code>
```

DESCRIPTION

This module is used to load byte compiled perl code. It uses the source filter mechanism to read the byte code and insert it into the compiled code at the appropriate point.

AUTHOR

Tom Hughes <tom@compton.nu> based on the ideas of Tim Bunce and others.

SEE ALSO

perl(1).

NAME

bytes – Perl pragma to force byte semantics rather than character semantics

SYNOPSIS

```
use bytes;
no bytes;
```

DESCRIPTION

The `use bytes` pragma disables character semantics for the rest of the lexical scope in which it appears. `no bytes` can be used to reverse the effect of `use bytes` within the current lexical scope.

Perl normally assumes character semantics in the presence of character data (i.e. data that has come from a source that has been marked as being of a particular character encoding). When `use bytes` is in effect, the encoding is temporarily ignored, and each string is treated as a series of bytes.

As an example, when Perl sees `$x = chr(400)`, it encodes the character in UTF-8 and stores it in `$x`. Then it is marked as character data, so, for instance, `length $x` returns 1. However, in the scope of the `bytes` pragma, `$x` is treated as a series of bytes – the bytes that make up the UTF8 encoding – and `length $x` returns 2:

```
$x = chr(400);
print "Length is ", length $x, "\n";      # "Length is 1"
printf "Contents are %vd\n", $x;          # "Contents are 400"
{
    use bytes;
    print "Length is ", length $x, "\n";  # "Length is 2"
    printf "Contents are %vd\n", $x;      # "Contents are 198.144"
}
```

For more on the implications and differences between character semantics and byte semantics, see [perlunicode](#).

SEE ALSO

[perlunicode](#), [utf8](#)

NAME

`carp` – warn of errors (from perspective of caller)

`cluck` – warn of errors with stack backtrace
(not exported by default)

`croak` – die of errors (from perspective of caller)

`confess` – die of errors with stack backtrace

`shortmess` – return the message that `carp` and `croak` produce

`longmess` – return the message that `cluck` and `confess` produce

SYNOPSIS

```
use Carp;
croak "We're outta here!";

use Carp qw(cluck);
cluck "This is how we got here!";

print FH Carp::shortmess("This will have caller's details added");
print FH Carp::longmess("This will have stack backtrace added");
```

DESCRIPTION

The Carp routines are useful in your own modules because they act like `die()` or `warn()`, but with a message which is more likely to be useful to a user of your module. In the case of `cluck`, `confess`, and `longmess` that context is a summary of every call in the call-stack. For a shorter message you can use `carp`, `croak` or `shortmess` which report the error as being from where your module was called. There is no guarantee that that is where the error was, but it is a good educated guess.

Here is a more complete description of how `shortmess` works. What it does is search the call-stack for a function call stack where it hasn't been told that there shouldn't be an error. If every call is marked safe, it then gives up and gives a full stack backtrace instead. In other words it presumes that the first likely looking potential suspect is guilty. Its rules for telling whether a call shouldn't generate errors work as follows:

1. Any call from a package to itself is safe.
2. Packages claim that there won't be errors on calls to or from packages explicitly marked as safe by inclusion in `@CARP_NOT`, or (if that array is empty) `@ISA`. The ability to override what `@ISA` says is new in 5.8.
3. The trust in item 2 is transitive. If A trusts B, and B trusts C, then A trusts C. So if you do not override `@ISA` with `@CARP_NOT`, then this trust relationship is identical to, "inherits from".
4. Any call from an internal Perl module is safe. (Nothing keeps user modules from marking themselves as internal to Perl, but this practice is discouraged.)
5. Any call to Carp is safe. (This rule is what keeps it from reporting the error where you call `carp/croak/shortmess`.)

Forcing a Stack Trace

As a debugging aid, you can force Carp to treat a `croak` as a `confess` and a `carp` as a `cluck` across *all* modules. In other words, force a detailed stack trace to be given. This can be very helpful when trying to understand why, or from where, a warning or error is being generated.

This feature is enabled by 'importing' the non-existent symbol 'verbose'. You would typically enable it by saying

```
perl -MCarp=verbose script.pl
```

or by including the string `MCarp=verbose` in the `PERL5OPT` environment variable.

BUGS

The Carp routines don't handle exception objects currently. If called with a first argument that is a reference, they simply call `die()` or `warn()`, as appropriate.

NAME

Carp heavy machinery – no user serviceable parts inside

NAME

CGI – Simple Common Gateway Interface Class

SYNOPSIS

```
# CGI script that creates a fill-out form
# and echoes back its values.

use CGI qw/:standard/;
print header,
      start_html('A Simple Example'),
      h1('A Simple Example'),
      start_form,
      "What's your name? ",textfield('name'),p,
      "What's the combination?", p,
      checkbox_group(-name=>'words',
                     -values=>['eenie','meenie','minie','moe'],
                     -defaults=>['eenie','minie']), p,
      "What's your favorite color? ",
      popup_menu(-name=>'color',
                 -values=>['red','green','blue','chartreuse']),p,
      submit,
      end_form,
      hr;

if (param()) {
    print "Your name is",em(param('name')),p,
          "The keywords are: ",em(join(" ",param('words'))),p,
          "Your favorite color is ",em(param('color')),
          hr;
}
```

ABSTRACT

This perl library uses perl5 objects to make it easy to create Web fill-out forms and parse their contents. This package defines CGI objects, entities that contain the values of the current query string and other state variables. Using a CGI object's methods, you can examine keywords and parameters passed to your script, and create forms whose initial values are taken from the current query (thereby preserving state information). The module provides shortcut functions that produce boilerplate HTML, reducing typing and coding errors. It also provides functionality for some of the more advanced features of CGI scripting, including support for file uploads, cookies, cascading style sheets, server push, and frames.

CGI.pm also provides a simple function-oriented programming style for those who don't need its object-oriented features.

The current version of CGI.pm is available at

```
http://www.genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html
ftp://ftp-genome.wi.mit.edu/pub/software/WWW/
```

DESCRIPTION**PROGRAMMING STYLE**

There are two styles of programming with CGI.pm, an object-oriented style and a function-oriented style. In the object-oriented style you create one or more CGI objects and then use object methods to create the various elements of the page. Each CGI object starts out with the list of named parameters that were passed to your CGI script by the server. You can modify the objects, save them to a file or database and recreate them. Because each object corresponds to the "state" of the CGI script, and because each object's parameter list is independent of the others, this allows you to save the state of the script and restore it later.

For example, using the object oriented style, here is how you create a simple "Hello World" HTML page:

```
#!/usr/local/bin/perl -w
use CGI;                               # load CGI routines
$q = new CGI;                          # create new CGI object
print $q->header,                      # create the HTTP header
      $q->start_html('hello world'), # start the HTML
      $q->h1('hello world'),         # level 1 header
      $q->end_html;                  # end the HTML
```

In the function-oriented style, there is one default CGI object that you rarely deal with directly. Instead you just call functions to retrieve CGI parameters, create HTML tags, manage cookies, and so on. This provides you with a cleaner programming interface, but limits you to using one CGI object at a time. The following example prints the same page, but uses the function-oriented interface. The main differences are that we now need to import a set of functions into our name space (usually the "standard" functions), and we don't need to create the CGI object.

```
#!/usr/local/bin/perl
use CGI qw/:standard/;                # load standard CGI routines
print header,                         # create the HTTP header
      start_html('hello world'), # start the HTML
      h1('hello world'),         # level 1 header
      end_html;                  # end the HTML
```

The examples in this document mainly use the object-oriented style. See [HOW TO IMPORT FUNCTIONS](#) for important information on function-oriented programming in CGI.pm

CALLING CGI.PM ROUTINES

Most CGI.pm routines accept several arguments, sometimes as many as 20 optional ones! To simplify this interface, all routines use a named argument calling style that looks like this:

```
print $q->header(-type=>'image/gif',-expires=>'+3d');
```

Each argument name is preceded by a dash. Neither case nor order matters in the argument list. `-type`, `-Type`, and `-TYPE` are all acceptable. In fact, only the first argument needs to begin with a dash. If a dash is present in the first argument, CGI.pm assumes dashes for the subsequent ones.

Several routines are commonly called with just one argument. In the case of these routines you can provide the single argument without an argument name. `header()` happens to be one of these routines. In this case, the single argument is the document type.

```
print $q->header('text/html');
```

Other such routines are documented below.

Sometimes named arguments expect a scalar, sometimes a reference to an array, and sometimes a reference to a hash. Often, you can pass any type of argument and the routine will do whatever is most appropriate. For example, the `param()` routine is used to set a CGI parameter to a single or a multi-valued value. The two cases are shown below:

```
$q->param(-name=>'veggie',-value=>'tomato');
$q->param(-name=>'veggie',-value=>['tomato','tomahto','potato','potahto']);
```

A large number of routines in CGI.pm actually aren't specifically defined in the module, but are generated automatically as needed. These are the "HTML shortcuts," routines that generate HTML tags for use in dynamically-generated pages. HTML tags have both attributes (the `attribute="value"` pairs within the tag itself) and contents (the part between the opening and closing pairs.) To distinguish between attributes and contents, CGI.pm uses the convention of passing HTML attributes as a hash reference as the first argument, and the contents, if any, as any subsequent arguments. It works out like this:

Code

Generated HTML

```

-----
h1()                                <h1>
h1('some','contents');             <h1>some contents</h1>
h1({-align=>left});                 <h1 ALIGN="LEFT">
h1({-align=>left},'contents');      <h1 ALIGN="LEFT">contents</h1>

```

HTML tags are described in more detail later.

Many newcomers to CGI.pm are puzzled by the difference between the calling conventions for the HTML shortcuts, which require curly braces around the HTML tag attributes, and the calling conventions for other routines, which manage to generate attributes without the curly brackets. Don't be confused. As a convenience the curly braces are optional in all but the HTML shortcuts. If you like, you can use curly braces when calling any routine that takes named arguments. For example:

```
print $q->header( {-type=>'image/gif',-expires=>'+3d'} );
```

If you use the `-w` switch, you will be warned that some CGI.pm argument names conflict with built-in Perl functions. The most frequent of these is the `-values` argument, used to create multi-valued menus, radio button clusters and the like. To get around this warning, you have several choices:

1. Use another name for the argument, if one is available. For example, `-value` is an alias for `-values`.
2. Change the capitalization, e.g. `-Values`
3. Put quotes around the argument name, e.g. `'-values'`

Many routines will do something useful with a named argument that it doesn't recognize. For example, you can produce non-standard HTTP header fields by providing them as named arguments:

```
print $q->header(-type => 'text/html',
                -cost  => 'Three smackers',
                -annoyance_level => 'high',
                -complaints_to   => 'bit bucket');
```

This will produce the following nonstandard HTTP header:

```
HTTP/1.0 200 OK
Cost: Three smackers
Annoyance-level: high
Complaints-to: bit bucket
Content-type: text/html
```

Notice the way that underscores are translated automatically into hyphens. HTML-generating routines perform a different type of translation.

This feature allows you to keep up with the rapidly changing HTTP and HTML "standards".

CREATING A NEW QUERY OBJECT (OBJECT-ORIENTED STYLE):

```
$query = new CGI;
```

This will parse the input (from both POST and GET methods) and store it into a perl5 object called `$query`.

CREATING A NEW QUERY OBJECT FROM AN INPUT FILE

```
$query = new CGI(INPUTFILE);
```

If you provide a file handle to the `new()` method, it will read parameters from the file (or STDIN, or whatever). The file can be in any of the forms describing below under debugging (i.e. a series of newline delimited TAG=VALUE pairs will work). Conveniently, this type of file is created by the `save()` method (see below). Multiple records can be saved and restored.

Perl purists will be pleased to know that this syntax accepts references to file handles, or even references to filehandle globs, which is the "official" way to pass a filehandle:

```
$query = new CGI(\*STDIN);
```

You can also initialize the CGI object with a FileHandle or IO::File object.

If you are using the function-oriented interface and want to initialize CGI state from a file handle, the way to do this is with `restore_parameters()`. This will (re)initialize the default CGI object from the indicated file handle.

```
open (IN,"test.in") || die;
restore_parameters(IN);
close IN;
```

You can also initialize the query object from an associative array reference:

```
$query = new CGI( { 'dinosaur'=>'barney',
                   'song'=>'I love you',
                   'friends'=>[qw/Jessica George Nancy/] }
);
```

or from a properly formatted, URL-escaped query string:

```
$query = new CGI('dinosaur=barney&color=purple');
```

or from a previously existing CGI object (currently this clones the parameter list, but none of the other object-specific fields, such as autoescaping):

```
$old_query = new CGI;
$new_query = new CGI($old_query);
```

To create an empty query, initialize it from an empty string or hash:

```
$empty_query = new CGI("");
```

-or-

```
$empty_query = new CGI({});
```

FETCHING A LIST OF KEYWORDS FROM THE QUERY:

```
@keywords = $query->keywords
```

If the script was invoked as the result of an <ISINDEX search, the parsed keywords can be obtained as an array using the `keywords()` method.

FETCHING THE NAMES OF ALL THE PARAMETERS PASSED TO YOUR SCRIPT:

```
@names = $query->param
```

If the script was invoked with a parameter list (e.g. "name1=value1&name2=value2&name3=value3"), the `param()` method will return the parameter names as a list. If the script was invoked as an <ISINDEX script and contains a string without ampersands (e.g. "value1+value2+value3"), there will be a single parameter named "keywords" containing the "+"-delimited keywords.

NOTE: As of version 1.5, the array of parameter names returned will be in the same order as they were submitted by the browser. Usually this order is the same as the order in which the parameters are defined in the form (however, this isn't part of the spec, and so isn't guaranteed).

FETCHING THE VALUE OR VALUES OF A SINGLE NAMED PARAMETER:

```
@values = $query->param('foo');
```

-or-

```
$value = $query->param('foo');
```

Pass the `param()` method a single argument to fetch the value of the named parameter. If the parameter is multivalued (e.g. from multiple selections in a scrolling list), you can ask to receive an array. Otherwise the

method will return a single value.

If a value is not given in the query string, as in the queries "name1=&name2=" or "name1&name2", it will be returned as an empty string. This feature is new in 2.63.

SETTING THE VALUE(S) OF A NAMED PARAMETER:

```
$query->param('foo','an','array','of','values');
```

This sets the value for the named parameter 'foo' to an array of values. This is one way to change the value of a field AFTER the script has been invoked once before. (Another way is with the `--override` parameter accepted by all methods that generate form elements.)

`param()` also recognizes a named parameter style of calling described in more detail later:

```
$query->param(-name=>'foo',-values=>['an','array','of','values']);
```

-or-

```
$query->param(-name=>'foo',-value=>'the value');
```

APPENDING ADDITIONAL VALUES TO A NAMED PARAMETER:

```
$query->append(-name=>'foo',-values=>['yet','more','values']);
```

This adds a value or list of values to the named parameter. The values are appended to the end of the parameter if it already exists. Otherwise the parameter is created. Note that this method only recognizes the named argument calling syntax.

IMPORTING ALL PARAMETERS INTO A NAMESPACE:

```
$query->import_names('R');
```

This creates a series of variables in the 'R' namespace. For example, `$R::foo`, `@R:foo`. For keyword lists, a variable `@R::keywords` will appear. If no namespace is given, this method will assume 'Q'. WARNING: don't import anything into 'main'; this is a major security risk!!!!

In older versions, this method was called `import()`. As of version 2.20, this name has been removed completely to avoid conflict with the built-in Perl module `import` operator.

DELETING A PARAMETER COMPLETELY:

```
$query->delete('foo');
```

This completely clears a parameter. It sometimes useful for resetting parameters that you don't want passed down between script invocations.

If you are using the function call interface, use `Delete()` instead to avoid conflicts with Perl's built-in `delete` operator.

DELETING ALL PARAMETERS:

```
$query->delete_all();
```

This clears the CGI object completely. It might be useful to ensure that all the defaults are taken when you create a fill-out form.

Use `Delete_all()` instead if you are using the function call interface.

DIRECT ACCESS TO THE PARAMETER LIST:

```
$q->param_fetch('address')->[1] = '1313 Mockingbird Lane';
unshift @{$q->param_fetch(-name=>'address')}, 'George Munster';
```

If you need access to the parameter list in a way that isn't covered by the methods above, you can obtain a direct reference to it by calling the `param_fetch()` method with the name of the . This will return an array reference to the named parameters, which you then can manipulate in any way you like.

You can also use a named argument style using the `--name` argument.

FETCHING THE PARAMETER LIST AS A HASH:

```
$params = $q->Vars;
print $params->{'address'};
@foo = split("\0", $params->{'foo'});
$params = $q->Vars;

use CGI ':cgi-lib';
$params = Vars;
```

Many people want to fetch the entire parameter list as a hash in which the keys are the names of the CGI parameters, and the values are the parameters' values. The `Vars()` method does this. Called in a scalar context, it returns the parameter list as a tied hash reference. Changing a key changes the value of the parameter in the underlying CGI parameter list. Called in a list context, it returns the parameter list as an ordinary hash. This allows you to read the contents of the parameter list, but not to change it.

When using this, the thing you must watch out for are multivalued CGI parameters. Because a hash cannot distinguish between scalar and list context, multivalued parameters will be returned as a packed string, separated by the `"\0"` (null) character. You must split this packed string in order to get at the individual values. This is the convention introduced long ago by Steve Brenner in his `cgi-lib.pl` module for Perl version 4.

If you wish to use `Vars()` as a function, import the `:cgi-lib` set of function calls (also see the section on CGI-LIB compatibility).

SAVING THE STATE OF THE SCRIPT TO A FILE:

```
$query->save(FILEHANDLE)
```

This will write the current state of the form to the provided filehandle. You can read it back in by providing a filehandle to the `new()` method. Note that the filehandle can be a file, a pipe, or whatever!

The format of the saved file is:

```
NAME1=VALUE1
NAME1=VALUE1'
NAME2=VALUE2
NAME3=VALUE3
=
```

Both name and value are URL escaped. Multi-valued CGI parameters are represented as repeated names. A session record is delimited by a single `=` symbol. You can write out multiple records and read them back in with several calls to `new`. You can do this across several sessions by opening the file in append mode, allowing you to create primitive guest books, or to keep a history of users' queries. Here's a short example of creating multiple session records:

```
use CGI;

open (OUT, ">>test.out") || die;
$records = 5;
foreach (0..$records) {
    my $q = new CGI;
    $q->param(-name=>'counter', -value=>$_);
    $q->save(OUT);
}
close OUT;

# reopen for reading
open (IN, "test.out") || die;
while (!eof(IN)) {
    my $q = new CGI(IN);
    print $q->param('counter'), "\n";
}
```

```
}
```

The file format used for save/restore is identical to that used by the Whitehead Genome Center's data exchange format "Boulderio", and can be manipulated and even databased using Boulderio utilities. See

<http://stein.cshl.org/boulder/>

for further details.

If you wish to use this method from the function-oriented (non-OO) interface, the exported name for this method is `save_parameters()`.

RETRIEVING CGI ERRORS

Errors can occur while processing user input, particularly when processing uploaded files. When these errors occur, CGI will stop processing and return an empty parameter list. You can test for the existence and nature of errors using the `cgi_error()` function. The error messages are formatted as HTTP status codes. You can either incorporate the error text into an HTML page, or use it as the value of the HTTP status:

```
my $error = $q->cgi_error;
if ($error) {
    print $q->header(-status=>$error),
          $q->start_html('Problems'),
          $q->h2('Request not processed'),
          $q->strong($error);
    exit 0;
}
```

When using the function-oriented interface (see the next section), errors may only occur the first time you call `param()`. Be ready for this!

USING THE FUNCTION-ORIENTED INTERFACE

To use the function-oriented interface, you must specify which CGI.pm routines or sets of routines to import into your script's namespace. There is a small overhead associated with this importation, but it isn't much.

```
use CGI <list of methods>;
```

The listed methods will be imported into the current package; you can call them directly without creating a CGI object first. This example shows how to import the `param()` and `header()` methods, and then use them directly:

```
use CGI 'param', 'header';
print header('text/plain');
$zipcode = param('zipcode');
```

More frequently, you'll import common sets of functions by referring to the groups by name. All function sets are preceded with a ":" character as in ":html3" (for tags defined in the HTML 3 standard).

Here is a list of the function sets you can import:

:cgi Import all CGI-handling methods, such as `param()`, `path_info()` and the like.

:form

Import all fill-out form generating methods, such as `textfield()`.

:html2

Import all methods that generate HTML 2.0 standard elements.

:html3

Import all methods that generate HTML 3.0 elements (such as `<table>`, `<sup>` and `<sub>`).

:html4

Import all methods that generate HTML 4 elements (such as <abbrev, <acronym and <thead).

:netscape

Import all methods that generate Netscape-specific HTML extensions.

:html

Import all HTML-generating shortcuts (i.e. 'html2' + 'html3' + 'netscape')...

:standard

Import "standard" features, 'html2', 'html3', 'html4', 'form' and 'cgi'.

:all Import all the available methods. For the full list, see the CGI.pm code, where the variable %EXPORT_TAGS is defined.

If you import a function name that is not part of CGI.pm, the module will treat it as a new HTML tag and generate the appropriate subroutine. You can then use it like any other HTML tag. This is to provide for the rapidly-evolving HTML "standard." For example, say Microsoft comes out with a new tag called <gradient (which causes the user's desktop to be flooded with a rotating gradient fill until his machine reboots). You don't need to wait for a new version of CGI.pm to start using it immediately:

```
use CGI qw/:standard :html3 gradient/;
print gradient({-start=>'red',-end=>'blue'});
```

Note that in the interests of execution speed CGI.pm does **not** use the standard *Exporter* syntax for specifying load symbols. This may change in the future.

If you import any of the state-maintaining CGI or form-generating methods, a default CGI object will be created and initialized automatically the first time you use any of the methods that require one to be present. This includes **param()**, **textfield()**, **submit()** and the like. (If you need direct access to the CGI object, you can find it in the global variable **\$CGI:Q**). By importing CGI.pm methods, you can create visually elegant scripts:

```
use CGI qw/:standard/;
print
    header,
    start_html('Simple Script'),
    h1('Simple Script'),
    start_form,
    "What's your name? ",textfield('name'),p,
    "What's the combination?",
    checkbox_group(-name=>'words',
                  -values=>['eenie','meenie','minie','moe'],
                  -defaults=>['eenie','moe']),p,
    "What's your favorite color?",
    popup_menu(-name=>'color',
               -values=>['red','green','blue','chartreuse']),p,
    submit,
    end_form,
    hr,"\n";
if (param) {
    print
        "Your name is ",em(param('name')),p,
        "The keywords are: ",em(join(" ",param('words'))),p,
        "Your favorite color is ",em(param('color')),".\n";
}
print end_html;
```

PRAGMAS

In addition to the function sets, there are a number of pragmas that you can import. Pragmas, which are always preceded by a hyphen, change the way that CGI.pm functions in various ways. Pragmas, function sets, and individual functions can all be imported in the same `use()` line. For example, the following `use` statement imports the standard set of functions and enables debugging mode (`pragma -debug`):

```
use CGI qw/:standard -debug/;
```

The current list of pragmas is as follows:

`-any`

When you *use CGI -any*, then any method that the query object doesn't recognize will be interpreted as a new HTML tag. This allows you to support the next *ad hoc* Netscape or Microsoft HTML extension. This lets you go wild with new and unsupported tags:

```
use CGI qw(-any);
$q=new CGI;
print $q->gradient({speed=>'fast',start=>'red',end=>'blue'});
```

Since using `<citeany>/cite` causes any mistyped method name to be interpreted as an HTML tag, use it with care or not at all.

`-compile`

This causes the indicated autoloading methods to be compiled up front, rather than deferred to later. This is useful for scripts that run for an extended period of time under FastCGI or `mod_perl`, and for those destined to be crunched by Malcom Beattie's Perl compiler. Use it in conjunction with the methods or method families you plan to use.

```
use CGI qw(-compile :standard :html3);
```

or even

```
use CGI qw(-compile :all);
```

Note that using the `-compile` pragma in this way will always have the effect of importing the compiled functions into the current namespace. If you want to compile without importing use the `compile()` method instead (see below).

`-nosticky`

This makes CGI.pm not generating the hidden fields `.submit` and `.cgifields`. It is very useful if you don't want to have the hidden fields appear in the `querystring` in a GET method. For example, a search script generated this way will have a very nice url with search parameters for bookmarking.

`-no_undef_params`

This keeps CGI.pm from including `undef` params in the parameter list.

`-no_xhtml`

By default, CGI.pm versions 2.69 and higher emit XHTML (<http://www.w3.org/TR/xhtml1/>). The `-no_xhtml` pragma disables this feature. Thanks to Michalis Kabrianis <kabrianis@hellug.gr> for this feature.

`-nph`

This makes CGI.pm produce a header appropriate for an NPH (no parsed header) script. You may need to do other things as well to tell the server that the script is NPH. See the discussion of NPH scripts below.

`-newstyle_urls`

Separate the `name=value` pairs in CGI parameter query strings with semicolons rather than ampersands. For example:

```
?name=fred;age=24;favorite_color=3
```

Semicolon-delimited query strings are always accepted, but will not be emitted by `self_url()` and `query_string()` unless the `-newstyle_urls` pragma is specified.

This became the default in version 2.64.

`-oldstyle_urls`

Separate the `name=value` pairs in CGI parameter query strings with ampersands rather than semicolons. This is no longer the default.

`-autoload`

This overrides the autoloader so that any function in your program that is not recognized is referred to CGI.pm for possible evaluation. This allows you to use all the CGI.pm functions without adding them to your symbol table, which is of concern for `mod_perl` users who are worried about memory consumption. *Warning:* when `-autoload` is in effect, you cannot use "poetry mode" (functions without the parenthesis). Use `hr()` rather than `hr`, or add something like `use subs qw/hr p header/` to the top of your script.

`-no_debug`

This turns off the command-line processing features. If you want to run a CGI.pm script from the command line to produce HTML, and you don't want it to read CGI parameters from the command line or STDIN, then use this pragma:

```
use CGI qw(-no_debug :standard);
```

`-debug`

This turns on full debugging. In addition to reading CGI arguments from the command-line processing, CGI.pm will pause and try to read arguments from STDIN, producing the message "(offline mode: enter name=value pairs on standard input)" features.

See the section on debugging for more details.

`-private_tempfiles`

CGI.pm can process uploaded file. Ordinarily it spools the uploaded file to a temporary directory, then deletes the file when done. However, this opens the risk of eavesdropping as described in the file upload section. Another CGI script author could peek at this data during the upload, even if it is confidential information. On Unix systems, the `-private_tempfiles` pragma will cause the temporary file to be unlinked as soon as it is opened and before any data is written into it, reducing, but not eliminating the risk of eavesdropping (there is still a potential race condition). To make life harder for the attacker, the program chooses tempfile names by calculating a 32 bit checksum of the incoming HTTP headers.

To ensure that the temporary file cannot be read by other CGI scripts, use `suEXEC` or a CGI wrapper program to run your script. The temporary file is created with mode 0600 (neither world nor group readable).

The temporary directory is selected using the following algorithm:

1. if the current user (e.g. "nobody") has a directory named "tmp" in its home directory, use that (Unix systems only).
2. if the environment variable `TMPDIR` exists, use the location indicated.
3. Otherwise try the locations `/usr/tmp`, `/var/tmp`, `C:\temp`, `/tmp`, `/temp`, `::Temporary Items`, and `\WWW_ROOT`.

Each of these locations is checked that it is a directory and is writable. If not, the algorithm tries the next choice.

SPECIAL FORMS FOR IMPORTING HTML-TAG FUNCTIONS

Many of the methods generate HTML tags. As described below, tag functions automatically generate both the opening and closing tags. For example:

```
print h1('Level 1 Header');
```

produces

```
<h1>Level 1 Header</h1>
```

There will be some times when you want to produce the start and end tags yourself. In this case, you can use the form `start_tag_name` and `end_tag_name`, as in:

```
print start_h1, 'Level 1 Header', end_h1;
```

With a few exceptions (described below), `start_tag_name` and `end_tag_name` functions are not generated automatically when you *use CGI*. However, you can specify the tags you want to generate *start/end* functions for by putting an asterisk in front of their name, or, alternatively, requesting either "`start_tag_name`" or "`end_tag_name`" in the import list.

Example:

```
use CGI qw/:standard *table start_ul/;
```

In this example, the following functions are generated in addition to the standard ones:

1. `start_table()` (generates a `<table` tag)
2. `end_table()` (generates a `</table` tag)
3. `start_ul()` (generates a `<ul` tag)
4. `end_ul()` (generates a `</ul` tag)

GENERATING DYNAMIC DOCUMENTS

Most of CGI.pm's functions deal with creating documents on the fly. Generally you will produce the HTTP header first, followed by the document itself. CGI.pm provides functions for generating HTTP headers of various types as well as for generating HTML. For creating GIF images, see the GD.pm module.

Each of these functions produces a fragment of HTML or HTTP which you can print out directly so that it displays in the browser window, append to a string, or save to a file for later use.

CREATING A STANDARD HTTP HEADER:

Normally the first thing you will do in any CGI script is print out an HTTP header. This tells the browser what type of document to expect, and gives other optional information, such as the language, expiration date, and whether to cache the document. The header can also be manipulated for special purposes, such as server push and pay per view pages.

```
print $query->header;
-or-
print $query->header('image/gif');
-or-
print $query->header('text/html', '204 No response');
-or-
print $query->header(-type=>'image/gif',
                    -nph=>1,
                    -status=>'402 Payment required',
                    -expires=>' +3d',
                    -cookie=>$cookie,
                    -charset=>'utf-7',
```

```
-attachment=>'foo.gif',
-Cost=>'$2.00');
```

`header()` returns the Content-type: header. You can provide your own MIME type if you choose, otherwise it defaults to text/html. An optional second parameter specifies the status code and a human-readable message. For example, you can specify 204, "No response" to create a script that tells the browser to do nothing at all.

The last example shows the named argument style for passing arguments to the CGI methods using named parameters. Recognized parameters are **-type**, **-status**, **-expires**, and **-cookie**. Any other named parameters will be stripped of their initial hyphens and turned into header fields, allowing you to specify any HTTP header you desire. Internal underscores will be turned into hyphens:

```
print $query->header(-Content_length=>3002);
```

Most browsers will not cache the output from CGI scripts. Every time the browser reloads the page, the script is invoked anew. You can change this behavior with the **-expires** parameter. When you specify an absolute or relative expiration interval with this parameter, some browsers and proxy servers will cache the script's output until the indicated expiration date. The following forms are all valid for the **-expires** field:

+30s	30 seconds from now
+10m	ten minutes from now
+1h	one hour from now
-1d	yesterday (i.e. "ASAP!")
now	immediately
+3M	in three months
+10y	in ten years time
Thursday, 25-Apr-1999 00:40:33 GMT at the indicated time & date	

The **-cookie** parameter generates a header that tells the browser to provide a "magic cookie" during all subsequent transactions with your script. Netscape cookies have a special format that includes interesting attributes such as expiration time. Use the `cookie()` method to create and retrieve session cookies.

The **-nph** parameter, if set to a true value, will issue the correct headers to work with an NPH (no-parse-header) script. This is important to use with certain servers that expect all their scripts to be NPH.

The **-charset** parameter can be used to control the character set sent to the browser. If not provided, defaults to ISO-8859-1. As a side effect, this sets the `charset()` method as well.

The **-attachment** parameter can be used to turn the page into an attachment. Instead of displaying the page, some browsers will prompt the user to save it to disk. The value of the argument is the suggested name for the saved file. In order for this to work, you may have to set the **-type** to "application/octet-stream".

GENERATING A REDIRECTION HEADER

```
print $query->redirect('http://somewhere.else/in/movie/land');
```

Sometimes you don't want to produce a document yourself, but simply redirect the browser elsewhere, perhaps choosing a URL based on the time of day or the identity of the user.

The `redirect()` function redirects the browser to a different URL. If you use redirection like this, you should **not** print out a header as well.

One hint I can offer is that relative links may not work correctly when you generate a redirection to another document on your site. This is due to a well-intentioned optimization that some servers use. The solution to this is to use the full URL (including the http: part) of the document you are redirecting to.

You can also use named arguments:

```
print $query->redirect(-uri=>'http://somewhere.else/in/movie/land',
                    -nph=>1);
```

The `-nph` parameter, if set to a true value, will issue the correct headers to work with an NPH (no-parse-header) script. This is important to use with certain servers, such as Microsoft Internet Explorer, which expect all their scripts to be NPH.

CREATING THE HTML DOCUMENT HEADER

```
print $query->start_html(-title=>'Secrets of the Pyramids',
                        -author=>'fred@capricorn.org',
                        -base=>'true',
                        -target=>'_blank',
                        -meta=>{'keywords'=>'pharaoh secret mummy',
                               'copyright'=>'copyright 1996 King Tut'},
                        -style=>{'src'=>'/styles/style1.css'},
                        -BGCOLOR=>'blue');
```

After creating the HTTP header, most CGI scripts will start writing out an HTML document. The `start_html()` routine creates the top of the page, along with a lot of optional information that controls the page's appearance and behavior.

This method returns a canned HTML header and the opening `<body` tag. All parameters are optional. In the named parameter form, recognized parameters are `-title`, `-author`, `-base`, `-xbase`, `-dtd`, `-lang` and `-target` (see below for the explanation). Any additional parameters you provide, such as the Netscape unofficial `BGCOLOR` attribute, are added to the `<body` tag. Additional parameters must be preceded by a hyphen.

The argument `-xbase` allows you to provide an HREF for the `<base` tag different from the current location, as in

```
-xbase=>"http://home.mcom.com/"
```

All relative links will be interpreted relative to this tag.

The argument `-target` allows you to provide a default target frame for all the links and fill-out forms on the page. **This is a non-standard HTTP feature which only works with Netscape browsers!** See the Netscape documentation on frames for details of how to manipulate this.

```
-target=>"answer_window"
```

All relative links will be interpreted relative to this tag. You add arbitrary meta information to the header with the `-meta` argument. This argument expects a reference to an associative array containing name/value pairs of meta information. These will be turned into a series of header `<meta` tags that look something like this:

```
<meta name="keywords" content="pharaoh secret mummy">
<meta name="description" content="copyright 1996 King Tut">
```

To create an HTTP-EQUIV type of `<meta` tag, use `-head`, described below.

The `-style` argument is used to incorporate cascading stylesheets into your code. See the section on CASCADING STYLESHEETS for more information.

The `-lang` argument is used to incorporate a language attribute into the `<html` tag. The default if not specified is "en-US" for US English. For example:

```
print $q->start_html(-lang=>'fr-CA');
```

The `-encoding` argument can be used to specify the character set for XHTML. It defaults to iso-8859-1 if not specified.

You can place other arbitrary HTML elements to the `<head` section with the `-head` tag. For example, to place the rarely-used `<link` element in the head section, use this:

```
print start_html(-head=>Link({'-rel'=>'next',
                             '-href'=>'http://www.capricorn.com/s2.html'}));
```


To incorporate multiple HTML elements into the <head section, just pass an array reference:

```
print start_html(-head=>[
    Link({-rel=>'next',
          -href=>'http://www.capricorn.com/s2.html'}),
    Link({-rel=>'previous',
          -href=>'http://www.capricorn.com/s1.html'})
]);
```

And here's how to create an HTTP-EQUIV <meta tag:

```
print start_html(-head=>meta({-http_equiv => 'Content-Type',
                              -content    => 'text/html'}))
```

JAVASCRIPTING: The **-script**, **-noScript**, **-onLoad**, **-onMouseOver**, **-onMouseOut** and **-onUnload** parameters are used to add Netscape JavaScript calls to your pages. **-script** should point to a block of text containing JavaScript function definitions. This block will be placed within a <script block inside the HTML (not HTTP) header. The block is placed in the header in order to give your page a fighting chance of having all its JavaScript functions in place even if the user presses the stop button before the page has loaded completely. CGI.pm attempts to format the script in such a way that JavaScript-naïve browsers will not choke on the code: unfortunately there are some browsers, such as Chimera for Unix, that get confused by it nevertheless.

The **-onLoad** and **-onUnload** parameters point to fragments of JavaScript code to execute when the page is respectively opened and closed by the browser. Usually these parameters are calls to functions defined in the **-script** field:

```
$query = new CGI;
print $query->header;
$JSCRIPT=<<END;
// Ask a silly question
function riddle_me_this() {
    var r = prompt("What walks on four legs in the morning, " +
                  "two legs in the afternoon, " +
                  "and three legs in the evening?");
    response(r);
}
// Get a silly answer
function response(answer) {
    if (answer == "man")
        alert("Right you are!");
    else
        alert("Wrong!  Guess again.");
}
END
print $query->start_html(-title=>'The Riddle of the Sphinx',
                       -script=>$JSCRIPT);
```

Use the **-noScript** parameter to pass some HTML text that will be displayed on browsers that do not have JavaScript (or browsers where JavaScript is turned off).

Netscape 3.0 recognizes several attributes of the <script tag, including LANGUAGE and SRC. The latter is particularly interesting, as it allows you to keep the JavaScript code in a file or CGI script rather than cluttering up each page with the source. To use these attributes pass a HASH reference in the **-script** parameter containing one or more of **-language**, **-src**, or **-code**:

```
print $q->start_html(-title=>'The Riddle of the Sphinx',
                   -script=>{-language=>'JAVASCRIPT',
```

```

        -src=>'/javascript/sphinx.js'}
    );

    print $q->(-title=>'The Riddle of the Sphinx',
              -script=>{-language=>'PERLSCRIPT',
                       -code=>'print "hello world!\n;"})
    ;

```

A final feature allows you to incorporate multiple <script sections into the header. Just pass the list of script sections as an array reference. this allows you to specify different source files for different dialects of JavaScript. Example:

```

    print $q->start_html(-title=>'The Riddle of the Sphinx',
                        -script=>[
                            { -language => 'JavaScript1.0',
                              -src       => '/javascript/utilities10.js'
                            },
                            { -language => 'JavaScript1.1',
                              -src       => '/javascript/utilities11.js'
                            },
                            { -language => 'JavaScript1.2',
                              -src       => '/javascript/utilities12.js'
                            },
                            { -language => 'JavaScript28.2',
                              -src       => '/javascript/utilities219.js'
                            }
                        ]
    );

```

If this looks a bit extreme, take my advice and stick with straight CGI scripting.

See

<http://home.netscape.com/eng/mozilla/2.0/handbook/javascript/>

for more information about JavaScript.

The old-style positional parameters are as follows:

Parameters:

1. The title
2. The author's e-mail address (will create a <link rev="MADE" tag if present
3. A 'true' flag if you want to include a <base tag in the header. This helps resolve relative addresses to absolute ones when the document is moved, but makes the document hierarchy non-portable. Use with care!
- 4, 5, 6...

Any other parameters you want to include in the <body tag. This is a good place to put Netscape extensions, such as colors and wallpaper patterns.

ENDING THE HTML DOCUMENT:

```
print $query->end_html
```

This ends an HTML document by printing the </body></html tags.

CREATING A SELF-REFERENCING URL THAT PRESERVES STATE INFORMATION:

```

$myself = $query->self_url;
print q(<a href="$myself">I'm talking to myself.</a>);

```

`self_url()` will return a URL, that, when selected, will reinvoke this script with all its state information intact. This is most useful when you want to jump around within the document using internal anchors but you don't want to disrupt the current contents of the form(s). Something like this will do the trick.

```
$myself = $query->self_url;
print "<a href=$myself#table1>See table 1</a>";
print "<a href=$myself#table2>See table 2</a>";
print "<a href=$myself#yourself>See for yourself</a>";
```

If you want more control over what's returned, using the `url()` method instead.

You can also retrieve the unprocessed query string with `query_string()`:

```
$the_string = $query->query_string;
```

OBTAINING THE SCRIPT'S URL

```
$full_url      = $query->url();
$full_url      = $query->url(-full=>1); #alternative syntax
$relative_url  = $query->url(-relative=>1);
$absolute_url  = $query->url(-absolute=>1);
$url_with_path = $query->url(-path_info=>1);
$url_with_path_and_query = $query->url(-path_info=>1,-query=>1);
$netloc        = $query->url(-base => 1);
```

`url()` returns the script's URL in a variety of formats. Called without any arguments, it returns the full form of the URL, including host name and port number

```
http://your.host.com/path/to/script.cgi
```

You can modify this format with the following named arguments:

-absolute

If true, produce an absolute URL, e.g.

```
/path/to/script.cgi
```

-relative

Produce a relative URL. This is useful if you want to reinvoke your script with different parameters. For example:

```
script.cgi
```

-full

Produce the full URL, exactly as if called without any arguments. This overrides the `-relative` and `-absolute` arguments.

-path (-path_info)

Append the additional path information to the URL. This can be combined with `-full`, `-absolute` or `-relative`. `-path_info` is provided as a synonym.

-query (-query_string)

Append the query string to the URL. This can be combined with `-full`, `-absolute` or `-relative`. `-query_string` is provided as a synonym.

-base

Generate just the protocol and net location, as in `http://www.foo.com:8000`

MIXING POST AND URL PARAMETERS

```
$color = $query->url_param('color');
```

It is possible for a script to receive CGI parameters in the URL as well as in the fill-out form by creating a form that POSTs to a URL containing a query string (a "?" mark followed by arguments). The `param()`

method will always return the contents of the POSTed fill-out form, ignoring the URL's query string. To retrieve URL parameters, call the `url_param()` method. Use it in the same way as `param()`. The main difference is that it allows you to read the parameters, but not set them.

Under no circumstances will the contents of the URL query string interfere with similarly-named CGI parameters in POSTed forms. If you try to mix a URL query string with a form submitted with the GET method, the results will not be what you expect.

CREATING STANDARD HTML ELEMENTS:

CGI.pm defines general HTML shortcut methods for most, if not all of the HTML 3 and HTML 4 tags. HTML shortcuts are named after a single HTML element and return a fragment of HTML text that you can then print or manipulate as you like. Each shortcut returns a fragment of HTML code that you can append to a string, save to a file, or, most commonly, print out so that it displays in the browser window.

This example shows how to use the HTML methods:

```
$q = new CGI;
print $q->blockquote(
    "Many years ago on the island of",
    $q->a({href=>"http://crete.org/"}, "Crete"),
    "there lived a Minotaur named",
    $q->strong("Fred."),
),
    $q->hr;
```

This results in the following HTML code (extra newlines have been added for readability):

```
<blockquote>
Many years ago on the island of
<a href="http://crete.org/">Crete</a> there lived
a minotaur named <strong>Fred.</strong>
</blockquote>
<hr>
```

If you find the syntax for calling the HTML shortcuts awkward, you can import them into your namespace and dispense with the object syntax completely (see the next section for more details):

```
use CGI ':standard';
print blockquote(
    "Many years ago on the island of",
    a({href=>"http://crete.org/"}, "Crete"),
    "there lived a minotaur named",
    strong("Fred."),
),
    hr;
```

PROVIDING ARGUMENTS TO HTML SHORTCUTS

The HTML methods will accept zero, one or multiple arguments. If you provide no arguments, you get a single tag:

```
print hr;    # <hr>
```

If you provide one or more string arguments, they are concatenated together with spaces and placed between opening and closing tags:

```
print h1("Chapter", "1"); # <h1>Chapter 1</h1>
```

If the first argument is an associative array reference, then the keys and values of the associative array become the HTML tag's attributes:

```
print a({-href=>'fred.html', -target=>'_new'},
```

```
"Open a new frame");
```

```
<a href="fred.html",target="_new">Open a new frame</a>
```

You may dispense with the dashes in front of the attribute names if you prefer:

```
print img {src=>'fred.gif',align=>'LEFT'};


```

Sometimes an HTML tag attribute has no argument. For example, ordered lists can be marked as COMPACT. The syntax for this is an argument that points to an undef string:

```
print ol({compact=>undef},li('one'),li('two'),li('three'));
```

Prior to CGI.pm version 2.41, providing an empty ("") string as an attribute argument was the same as providing undef. However, this has changed in order to accommodate those who want to create tags of the form . The difference is shown in these two pieces of code:

CODE	RESULT
<code>img({alt=>undef})</code>	<code></code>
<code>img({alt=>''})</code>	<code></code>

THE DISTRIBUTIVE PROPERTY OF HTML SHORTCUTS

One of the cool features of the HTML shortcuts is that they are distributive. If you give them an argument consisting of a **reference** to a list, the tag will be distributed across each element of the list. For example, here's one way to make an ordered list:

```
print ul(
    li({-type=>'disc'}, ['Sneezy', 'Doc', 'Sleepy', 'Happy'])
);
```

This example will result in HTML output that looks like this:

```
<ul>
  <li type="disc">Sneezy</li>
  <li type="disc">Doc</li>
  <li type="disc">Sleepy</li>
  <li type="disc">Happy</li>
</ul>
```

This is extremely useful for creating tables. For example:

```
print table({-border=>undef},
    caption('When Should You Eat Your Vegetables?'),
    Tr({-align=>CENTER,-valign=>TOP},
    [
        th(['Vegetable', 'Breakfast', 'Lunch', 'Dinner']),
        td(['Tomatoes', 'no', 'yes', 'yes']),
        td(['Broccoli', 'no', 'no', 'yes']),
        td(['Onions', 'yes', 'yes', 'yes'])
    ]
    )
);
```

HTML SHORTCUTS AND LIST INTERPOLATION

Consider this bit of code:

```
print blockquote(em('Hi'), 'mom!');
```

It will ordinarily return the string that you probably expect, namely:

```
<blockquote><em>Hi</em> mom!</blockquote>
```

Note the space between the element "Hi" and the element "mom!". CGI.pm puts the extra space there using array interpolation, which is controlled by the magic `$"` variable. Sometimes this extra space is not what you want, for example, when you are trying to align a series of images. In this case, you can simply change the value of `$"` to an empty string.

```
{
    local($") = '';
    print blockquote(em('Hi'), 'mom!'));
}
```

I suggest you put the code in a block as shown here. Otherwise the change to `$"` will affect all subsequent code until you explicitly reset it.

NON-STANDARD HTML SHORTCUTS

A few HTML tags don't follow the standard pattern for various reasons.

`comment()` generates an HTML comment (`<!-- comment -->`). Call it like

```
print comment('here is my comment');
```

Because of conflicts with built-in Perl functions, the following functions begin with initial caps:

```
Select
Tr
Link
Delete
Accept
Sub
```

In addition, `start_html()`, `end_html()`, `start_form()`, `end_form()`, `start_multipart_form()` and all the fill-out form tags are special. See their respective sections.

AUTOESCAPING HTML

By default, all HTML that is emitted by the form-generating functions is passed through a function called `escapeHTML()`:

```
$escaped_string = escapeHTML("unescaped string");
```

Escape HTML formatting characters in a string.

Provided that you have specified a character set of ISO-8859-1 (the default), the standard HTML escaping rules will be used. The "<" character becomes "<", ">" becomes ">", "&" becomes "&", and the quote character becomes """. In addition, the hexadecimal 0x8b and 0x9b characters, which many windows-based browsers interpret as the left and right angle-bracket characters, are replaced by their numeric HTML entities ("‹" and "›"). If you manually change the charset, either by calling the `charset()` method explicitly or by passing a `-charset` argument to `header()`, then **all** characters will be replaced by their numeric entities, since CGI.pm has no lookup table for all the possible encodings.

The automatic escaping does not apply to other shortcuts, such as `h1()`. You should call `escapeHTML()` yourself on untrusted data in order to protect your pages against nasty tricks that people may enter into guestbooks, etc.. To change the character set, use `charset()`. To turn autoescaping off completely, use `autoescape()`:

```
$charset = charset([$charset]);
```

Get or set the current character set.

```
$flag = autoEscape([$flag]);
```

Get or set the value of the autoescape flag.

PRETTY-PRINTING HTML

By default, all the HTML produced by these functions comes out as one long line without carriage returns or indentation. This is yuck, but it does reduce the size of the documents by 10–20%. To get pretty-printed output, please use *CGI::Pretty*, a subclass contributed by Brian Paulsen.

CREATING FILL-OUT FORMS:

General note The various form-creating methods all return strings to the caller, containing the tag or tags that will create the requested form element. You are responsible for actually printing out these strings. It's set up this way so that you can place formatting tags around the form elements.

Another note The default values that you specify for the forms are only used the **first** time the script is invoked (when there is no query string). On subsequent invocations of the script (when there is a query string), the former values are used even if they are blank.

If you want to change the value of a field from its previous value, you have two choices:

- (1) call the `param()` method to set it.
- (2) use the `-override` (alias `-force`) parameter (a new feature in version 2.15). This forces the default value to be used, regardless of the previous value:

```
print $query->textfield(-name=>'field_name',
                      -default=>'starting value',
                      -override=>1,
                      -size=>50,
                      -maxlength=>80);
```

Yet another note By default, the text and labels of form elements are escaped according to HTML rules. This means that you can safely use "<CLICK ME" as the label for a button. However, it also interferes with your ability to incorporate special HTML character sequences, such as `Á`, into your fields. If you wish to turn off automatic escaping, call the `autoEscape()` method with a false value immediately after creating the CGI object:

```
$query = new CGI;
$query->autoEscape(undef);
```

CREATING AN ISINDEX TAG

```
print $query->isindex(-action=>$action);

-or-

print $query->isindex($action);
```

Prints out an `<isindex` tag. Not very exciting. The parameter `-action` specifies the URL of the script to process the query. The default is to process the query with the current script.

STARTING AND ENDING A FORM

```
print $query->start_form(-method=>$method,
                      -action=>$action,
                      -enctype=>$encoding);
<... various form stuff ...>
print $query->endform;

-or-

print $query->start_form($method,$action,$encoding);
<... various form stuff ...>
print $query->endform;
```

`start_form()` will return a `<form` tag with the optional method, action and form encoding that you specify. The defaults are:

```
method: POST
action: this script
enctype: application/x-www-form-urlencoded
```

`endform()` returns the closing `</form` tag.

`Start_form()`'s `enctype` argument tells the browser how to package the various fields of the form before sending the form to the server. Two values are possible:

Note: This method was previously named `startform()`, and `startform()` is still recognized as an alias.

application/x-www-form-urlencoded

This is the older type of encoding used by all browsers prior to Netscape 2.0. It is compatible with many CGI scripts and is suitable for short fields containing text data. For your convenience, CGI.pm stores the name of this encoding type in `&CGI::URL_ENCODED`.

multipart/form-data

This is the newer type of encoding introduced by Netscape 2.0. It is suitable for forms that contain very large fields or that are intended for transferring binary data. Most importantly, it enables the "file upload" feature of Netscape 2.0 forms. For your convenience, CGI.pm stores the name of this encoding type in `&CGI::MULTIPART`.

Forms that use this type of encoding are not easily interpreted by CGI scripts unless they use CGI.pm or another library designed to handle them.

For compatibility, the `start_form()` method uses the older form of encoding by default. If you want to use the newer form of encoding by default, you can call `start_multipart_form()` instead of `start_form()`.

JAVASCRIPTING: The `-name` and `-onSubmit` parameters are provided for use with JavaScript. The `-name` parameter gives the form a name so that it can be identified and manipulated by JavaScript functions. `-onSubmit` should point to a JavaScript function that will be executed just before the form is submitted to your server. You can use this opportunity to check the contents of the form for consistency and completeness. If you find something wrong, you can put up an alert box or maybe fix things up yourself. You can abort the submission by returning false from this function.

Usually the bulk of JavaScript functions are defined in a `<script` block in the HTML header and `-onSubmit` points to one of these function call. See `start_html()` for details.

CREATING A TEXT FIELD

```
print $query->textfield(-name=>'field_name',
                        -default=>'starting value',
                        -size=>50,
                        -maxlength=>80);

-or-

print $query->textfield('field_name','starting value',50,80);
```

`textfield()` will return a text input field.

Parameters

1. The first parameter is the required name for the field (`-name`).
2. The optional second parameter is the default starting value for the field contents (`-default`).
3. The optional third parameter is the size of the field in characters (`-size`).
4. The optional fourth parameter is the maximum number of characters the field will accept (`-maxlength`).

As with all these methods, the field will be initialized with its previous contents from earlier invocations of the script. When the form is processed, the value of the text field can be retrieved with:

```
$value = $query->param('foo');
```

If you want to reset it from its initial value after the script has been called once, you can do so like this:

```
$query->param('foo',"I'm taking over this value!");
```

NEW AS OF VERSION 2.15: If you don't want the field to take on its previous value, you can force its current value by using the `-override` (alias `-force`) parameter:

```
print $query->textfield(-name=>'field_name',
                      -default=>'starting value',
                      -override=>1,
                      -size=>50,
                      -maxlength=>80);
```

JAVASCRIPTING: You can also provide `-onChange`, `-onFocus`, `-onBlur`, `-onMouseOver`, `-onMouseOut` and `-onSelect` parameters to register JavaScript event handlers. The `onChange` handler will be called whenever the user changes the contents of the text field. You can do text validation if you like. `onFocus` and `onBlur` are called respectively when the insertion point moves into and out of the text field. `onSelect` is called when the user changes the portion of the text that is selected.

CREATING A BIG TEXT FIELD

```
print $query->textarea(-name=>'foo',
                     -default=>'starting value',
                     -rows=>10,
                     -columns=>50);
```

-or-

```
print $query->textarea('foo','starting value',10,50);
```

`textarea()` is just like `textfield`, but it allows you to specify rows and columns for a multiline text entry box. You can provide a starting value for the field, which can be long and contain multiple lines.

JAVASCRIPTING: The `-onChange`, `-onFocus`, `-onBlur`, `-onMouseOver`, `-onMouseOut`, and `-onSelect` parameters are recognized. See `textfield()`.

CREATING A PASSWORD FIELD

```
print $query->password_field(-name=>'secret',
                           -value=>'starting value',
                           -size=>50,
                           -maxlength=>80);
```

-or-

```
print $query->password_field('secret','starting value',50,80);
```

`password_field()` is identical to `textfield()`, except that its contents will be starred out on the web page.

JAVASCRIPTING: The `-onChange`, `-onFocus`, `-onBlur`, `-onMouseOver`, `-onMouseOut` and `-onSelect` parameters are recognized. See `textfield()`.

CREATING A FILE UPLOAD FIELD

```
print $query->filefield(-name=>'uploaded_file',
                      -default=>'starting value',
                      -size=>50,
                      -maxlength=>80);
```

-or-

```
print $query->filefield('uploaded_file','starting value',50,80);
```

`filefield()` will return a file upload field for Netscape 2.0 browsers. In order to take full advantage of this *you must use the new multipart encoding scheme* for the form. You can do this either by calling `start_form()` with an encoding type of `&CGI::MULTIPART`, or by calling the new method `start_multipart_form()` instead of vanilla `start_form()`.

Parameters

1. The first parameter is the required name for the field (`--name`).
2. The optional second parameter is the starting value for the field contents to be used as the default file name (`--default`).

For security reasons, browsers don't pay any attention to this field, and so the starting value will always be blank. Worse, the field loses its "sticky" behavior and forgets its previous contents. The starting value field is called for in the HTML specification, however, and possibly some browser will eventually provide support for it.

3. The optional third parameter is the size of the field in characters (`--size`).
4. The optional fourth parameter is the maximum number of characters the field will accept (`--maxlength`).

When the form is processed, you can retrieve the entered filename by calling `param()`:

```
$filename = $query->param('uploaded_file');
```

Different browsers will return slightly different things for the name. Some browsers return the filename only. Others return the full path to the file, using the path conventions of the user's machine. Regardless, the name returned is always the name of the file on the *user's* machine, and is unrelated to the name of the temporary file that CGI.pm creates during upload spooling (see below).

The filename returned is also a file handle. You can read the contents of the file using standard Perl file reading calls:

```
# Read a text file and print it out
while (<$filename>) {
    print;
}

# Copy a binary file to somewhere safe
open (OUTFILE,">>/usr/local/web/users/feedback");
while ($bytesread=read($filename,$buffer,1024)) {
    print OUTFILE $buffer;
}
```

However, there are problems with the dual nature of the upload fields. If you use `strict`, then Perl will complain when you try to use a string as a filehandle. You can get around this by placing the file reading code in a block containing the `no strict` pragma. More seriously, it is possible for the remote user to type garbage into the upload field, in which case what you get from `param()` is not a filehandle at all, but a string.

To be safe, use the `upload()` function (new in version 2.47). When called with the name of an upload field, `upload()` returns a filehandle, or `undef` if the parameter is not a valid filehandle.

```
$fh = $query->upload('uploaded_file');
while (<$fh>) {
    print;
}
```

In an array context, `upload()` will return an array of filehandles. This makes it possible to create forms that use the same name for multiple upload fields.

This is the recommended idiom.

When a file is uploaded the browser usually sends along some information along with it in the format of headers. The information usually includes the MIME content type. Future browsers may send other information as well (such as modification date and size). To retrieve this information, call `uploadInfo()`. It returns a reference to an associative array containing all the document headers.

```
$filename = $query->param('uploaded_file');
$type = $query->uploadInfo($filename)->{'Content-Type'};
unless ($type eq 'text/html') {
    die "HTML FILES ONLY!";
}
```

If you are using a machine that recognizes "text" and "binary" data modes, be sure to understand when and how to use them (see the Camel book). Otherwise you may find that binary files are corrupted during file uploads.

There are occasionally problems involving parsing the uploaded file. This usually happens when the user presses "Stop" before the upload is finished. In this case, CGI.pm will return undef for the name of the uploaded file and set `cgi_error()` to the string "400 Bad request (malformed multipart POST)". This error message is designed so that you can incorporate it into a status code to be sent to the browser. Example:

```
$file = $query->upload('uploaded_file');
if (!$file && $query->cgi_error) {
    print $query->header(-status=>$query->cgi_error);
    exit 0;
}
```

You are free to create a custom HTML page to complain about the error, if you wish.

If you are using CGI.pm on a Windows platform and find that binary files get slightly larger when uploaded but that text files remain the same, then you have forgotten to activate binary mode on the output filehandle. Be sure to call `binmode()` on any handle that you create to write the uploaded file to disk.

JAVASCRIPTING: The `-onChange`, `-onFocus`, `-onBlur`, `-onmouseover`, `-onmouseout` and `-onselect` parameters are recognized. See `textfield()` for details.

CREATING A POPUP MENU

```
print $query->popup_menu('menu_name',
                        ['eenie', 'meenie', 'minie'],
                        'meenie');

-or-

%labels = ('eenie'=>'your first choice',
           'meenie'=>'your second choice',
           'minie'=>'your third choice');
print $query->popup_menu('menu_name',
                        ['eenie', 'meenie', 'minie'],
                        'meenie', \%labels);

-or (named parameter style)-

print $query->popup_menu(-name=>'menu_name',
                        -values=>['eenie', 'meenie', 'minie'],
                        -default=>'meenie',
                        -labels=>\%labels);
```

`popup_menu()` creates a menu.

1. The required first argument is the menu's name (`--name`).
2. The required second argument (`--values`) is an array **reference** containing the list of menu items in the menu. You can pass the method an anonymous array, as shown in the example, or a reference to a named array, such as `"\@foo"`.
3. The optional third parameter (`--default`) is the name of the default menu choice. If not specified, the first item will be the default. The values of the previous choice will be maintained across queries.
4. The optional fourth parameter (`--labels`) is provided for people who want to use different values for the user-visible label inside the popup menu and the value returned to your script. It's a pointer to an associative array relating menu values to user-visible labels. If you leave this parameter blank, the menu values will be displayed by default. (You can also leave a label undefined if you want to).

When the form is processed, the selected value of the popup menu can be retrieved using:

```
$popup_menu_value = $query->param('menu_name');
```

JAVASCRIPTING: `popup_menu()` recognizes the following event handlers: `--onChange`, `--onFocus`, `--onMouseOver`, `--onMouseOut`, and `--onBlur`. See the `textfield()` section for details on when these handlers are called.

CREATING A SCROLLING LIST

```
print $query->scrolling_list('list_name',
                             ['eenie', 'meenie', 'minie', 'moe'],
                             ['eenie', 'moe'], 5, 'true');
```

-or-

```
print $query->scrolling_list('list_name',
                             ['eenie', 'meenie', 'minie', 'moe'],
                             ['eenie', 'moe'], 5, 'true',
                             \%labels);
```

-or-

```
print $query->scrolling_list(-name=>'list_name',
                             -values=>['eenie', 'meenie', 'minie', 'moe'],
                             -default=>['eenie', 'moe'],
                             -size=>5,
                             -multiple=>'true',
                             -labels=>\%labels);
```

`scrolling_list()` creates a scrolling list.

Parameters:

1. The first and second arguments are the list name (`--name`) and values (`--values`). As in the popup menu, the second argument should be an array reference.
2. The optional third argument (`--default`) can be either a reference to a list containing the values to be selected by default, or can be a single value to select. If this argument is missing or undefined, then nothing is selected when the list first appears. In the named parameter version, you can use the synonym `--defaults` for this parameter.
3. The optional fourth argument is the size of the list (`--size`).
4. The optional fifth argument can be set to true to allow multiple simultaneous selections (`--multiple`). Otherwise only one selection will be allowed at a time.
5. The optional sixth argument is a pointer to an associative array containing long user-visible labels for the list items (`--labels`). If not provided, the values will be displayed.

When this form is processed, all selected list items will be returned as a list under the parameter name

'list_name'. The values of the selected items can be retrieved with:

```
@selected = $query->param('list_name');
```

JAVASCRIPTING: `scrolling_list()` recognizes the following event handlers: **-onChange**, **-onFocus**, **-onMouseOver**, **-onMouseOut** and **-onBlur**. See `textfield()` for the description of when these handlers are called.

CREATING A GROUP OF RELATED CHECKBOXES

```
print $query->checkbox_group(-name=>'group_name',
                          -values=>['eenie', 'meenie', 'minie', 'moe'],
                          -default=>['eenie', 'moe'],
                          -linebreak=>'true',
                          -labels=>%labels);
```

```
print $query->checkbox_group('group_name',
                          ['eenie', 'meenie', 'minie', 'moe'],
                          ['eenie', 'moe'], 'true', %labels);
```

HTML3-COMPATIBLE BROWSERS ONLY:

```
print $query->checkbox_group(-name=>'group_name',
                          -values=>['eenie', 'meenie', 'minie', 'moe'],
                          -rows=2, -columns=>2);
```

`checkbox_group()` creates a list of checkboxes that are related by the same name.

Parameters:

1. The first and second arguments are the checkbox name and values, respectively (`-name` and `-values`). As in the popup menu, the second argument should be an array reference. These values are used for the user-readable labels printed next to the checkboxes as well as for the values passed to your script in the query string.
2. The optional third argument (`-default`) can be either a reference to a list containing the values to be checked by default, or can be a single value to checked. If this argument is missing or undefined, then nothing is selected when the list first appears.
3. The optional fourth argument (`-linebreak`) can be set to true to place line breaks between the checkboxes so that they appear as a vertical list. Otherwise, they will be strung together on a horizontal line.
4. The optional fifth argument is a pointer to an associative array relating the checkbox values to the user-visible labels that will be printed next to them (`-labels`). If not provided, the values will be used as the default.
5. **HTML3-compatible browsers** (such as Netscape) can take advantage of the optional parameters `-rows`, and `-columns`. These parameters cause `checkbox_group()` to return an HTML3 compatible table containing the checkbox group formatted with the specified number of rows and columns. You can provide just the `-columns` parameter if you wish; `checkbox_group` will calculate the correct number of rows for you.

To include row and column headings in the returned table, you can use the `-rowheaders` and `-colheaders` parameters. Both of these accept a pointer to an array of headings to use. The headings are just decorative. They don't reorganize the interpretation of the checkboxes — they're still a single named unit.

When the form is processed, all checked boxes will be returned as a list under the parameter name 'group_name'. The values of the "on" checkboxes can be retrieved with:

```
@turned_on = $query->param('group_name');
```

The value returned by `checkbox_group()` is actually an array of button elements. You can capture them

and use them within tables, lists, or in other creative ways:

```
@h = $query->checkbox_group(-name=>'group_name',-values=>\@values);
&use_in_creative_way(@h);
```

JAVASCRIPTING: `checkbox_group()` recognizes the **-onClick** parameter. This specifies a JavaScript code fragment or function call to be executed every time the user clicks on any of the buttons in the group. You can retrieve the identity of the particular button clicked on using the "this" variable.

CREATING A STANDALONE CHECKBOX

```
print $query->checkbox(-name=>'checkbox_name',
                    -checked=>1,
                    -value=>'ON',
                    -label=>'CLICK ME');
```

-or-

```
print $query->checkbox('checkbox_name', 'checked', 'ON', 'CLICK ME');
```

`checkbox()` is used to create an isolated checkbox that isn't logically related to any others.

Parameters:

1. The first parameter is the required name for the checkbox (`-name`). It will also be used for the user-readable label printed next to the checkbox.
2. The optional second parameter (`-checked`) specifies that the checkbox is turned on by default. Synonyms are `-selected` and `-on`.
3. The optional third parameter (`-value`) specifies the value of the checkbox when it is checked. If not provided, the word "on" is assumed.
4. The optional fourth parameter (`-label`) is the user-readable label to be attached to the checkbox. If not provided, the checkbox name is used.

The value of the checkbox can be retrieved using:

```
$turned_on = $query->param('checkbox_name');
```

JAVASCRIPTING: `checkbox()` recognizes the **-onClick** parameter. See `checkbox_group()` for further details.

CREATING A RADIO BUTTON GROUP

```
print $query->radio_group(-name=>'group_name',
                         -values=>['eenie','meenie','minie'],
                         -default=>'meenie',
                         -linebreak=>'true',
                         -labels=>\%labels);
```

-or-

```
print $query->radio_group('group_name', ['eenie','meenie','minie'],
                         'meenie','true',\%labels);
```

HTML3-COMPATIBLE BROWSERS ONLY:

```
print $query->radio_group(-name=>'group_name',
                         -values=>['eenie','meenie','minie','moe'],
                         -rows=2,-columns=>2);
```

`radio_group()` creates a set of logically-related radio buttons (turning one member of the group on turns the others off)

Parameters:

1. The first argument is the name of the group and is required (`-name`).
2. The second argument (`-values`) is the list of values for the radio buttons. The values and the labels that appear on the page are identical. Pass an array *reference* in the second argument, either using an anonymous array, as shown, or by referencing a named array as in `"\@foo"`.
3. The optional third parameter (`-default`) is the name of the default button to turn on. If not specified, the first item will be the default. You can provide a nonexistent button name, such as `"-"` to start up with no buttons selected.
4. The optional fourth parameter (`-linebreak`) can be set to `'true'` to put line breaks between the buttons, creating a vertical list.
5. The optional fifth parameter (`-labels`) is a pointer to an associative array relating the radio button values to user-visible labels to be used in the display. If not provided, the values themselves are displayed.
6. **HTML3-compatible browsers** (such as Netscape) can take advantage of the optional parameters `-rows`, and `-columns`. These parameters cause `radio_group()` to return an HTML3 compatible table containing the radio group formatted with the specified number of rows and columns. You can provide just the `-columns` parameter if you wish; `radio_group` will calculate the correct number of rows for you.

To include row and column headings in the returned table, you can use the `-rowheader` and `-colheader` parameters. Both of these accept a pointer to an array of headings to use. The headings are just decorative. They don't reorganize the interpretation of the radio buttons — they're still a single named unit.

When the form is processed, the selected radio button can be retrieved using:

```
$which_radio_button = $query->param('group_name');
```

The value returned by `radio_group()` is actually an array of button elements. You can capture them and use them within tables, lists, or in other creative ways:

```
@h = $query->radio_group(-name=>'group_name', -values=>\@values);
use_in_creative_way(@h);
```

CREATING A SUBMIT BUTTON

```
print $query->submit(-name=>'button_name',
                   -value=>'value');
```

-or-

```
print $query->submit('button_name', 'value');
```

`submit()` will create the query submission button. Every form should have one of these.

Parameters:

1. The first argument (`-name`) is optional. You can give the button a name if you have several submission buttons in your form and you want to distinguish between them. The name will also be used as the user-visible label. Be aware that a few older browsers don't deal with this correctly and **never** send back a value from a button.
2. The second argument (`-value`) is also optional. This gives the button a value that will be passed to your script in the query string.

You can figure out which button was pressed by using different values for each one:

```
$which_one = $query->param('button_name');
```

JAVASCRIPTING: `radio_group()` recognizes the `-onClick` parameter. See `checkbox_group()`

for further details.

CREATING A RESET BUTTON

```
print $query->reset
```

`reset()` creates the "reset" button. Note that it restores the form to its value from the last time the script was called, NOT necessarily to the defaults.

Note that this conflicts with the Perl `reset()` built-in. Use `CORE::reset()` to get the original reset function.

CREATING A DEFAULT BUTTON

```
print $query->defaults('button_label')
```

`defaults()` creates a button that, when invoked, will cause the form to be completely reset to its defaults, wiping out all the changes the user ever made.

CREATING A HIDDEN FIELD

```
print $query->hidden(-name=>'hidden_name',  
                    -default=>['value1','value2'...]);
```

-or-

```
print $query->hidden('hidden_name','value1','value2'...);
```

`hidden()` produces a text field that can't be seen by the user. It is useful for passing state variable information from one invocation of the script to the next.

Parameters:

1. The first argument is required and specifies the name of this field (`-name`).
2. The second argument is also required and specifies its value (`-default`). In the named parameter style of calling, you can provide a single value here or a reference to a whole list

Fetch the value of a hidden field this way:

```
$hidden_value = $query->param('hidden_name');
```

Note, that just like all the other form elements, the value of a hidden field is "sticky". If you want to replace a hidden field with some other values after the script has been called once you'll have to do it manually:

```
$query->param('hidden_name','new','values','here');
```

CREATING A CLICKABLE IMAGE BUTTON

```
print $query->image_button(-name=>'button_name',  
                           -src=>'/source/URL',  
                           -align=>'MIDDLE');
```

-or-

```
print $query->image_button('button_name','/source/URL','MIDDLE');
```

`image_button()` produces a clickable image. When it's clicked on the position of the click is returned to your script as "button_name.x" and "button_name.y", where "button_name" is the name you've assigned to it.

JAVASCRIPTING: `image_button()` recognizes the `-onClick` parameter. See `checkbox_group()` for further details.

Parameters:

1. The first argument (`-name`) is required and specifies the name of this field.
2. The second argument (`-src`) is also required and specifies the URL

- The third option (`-align`, optional) is an alignment type, and may be TOP, BOTTOM or MIDDLE

Fetch the value of the button this way:

```
$x = $query-param('button_name.x');
$y = $query-param('button_name.y');
```

CREATING A JAVASCRIPT ACTION BUTTON

```
print $query->button(-name=>'button_name',
                    -value=>'user visible label',
                    -onClick=>"do_something()");
```

-or-

```
print $query->button('button_name', "do_something()");
```

`button()` produces a button that is compatible with Netscape 2.0's JavaScript. When it's pressed the fragment of JavaScript code pointed to by the `-onClick` parameter will be executed. On non-Netscape browsers this form element will probably not even display.

HTTP COOKIES

Netscape browsers versions 1.1 and higher, and all versions of Internet Explorer, support a so-called "cookie" designed to help maintain state within a browser session. CGI.pm has several methods that support cookies.

A cookie is a name=value pair much like the named parameters in a CGI query string. CGI scripts create one or more cookies and send them to the browser in the HTTP header. The browser maintains a list of cookies that belong to a particular Web server, and returns them to the CGI script during subsequent interactions.

In addition to the required name=value pair, each cookie has several optional attributes:

1. an expiration time

This is a time/date string (in a special GMT format) that indicates when a cookie expires. The cookie will be saved and returned to your script until this expiration date is reached if the user exits the browser and restarts it. If an expiration date isn't specified, the cookie will remain active until the user quits the browser.

2. a domain

This is a partial or complete domain name for which the cookie is valid. The browser will return the cookie to any host that matches the partial domain name. For example, if you specify a domain name of ".capricorn.com", then the browser will return the cookie to Web servers running on any of the machines "www.capricorn.com", "www2.capricorn.com", "feckless.capricorn.com", etc. Domain names must contain at least two periods to prevent attempts to match on top level domains like ".edu". If no domain is specified, then the browser will only return the cookie to servers on the host the cookie originated from.

3. a path

If you provide a cookie path attribute, the browser will check it against your script's URL before returning the cookie. For example, if you specify the path "/cgi-bin", then the cookie will be returned to each of the scripts "/cgi-bin/tally.pl", "/cgi-bin/order.pl", and "/cgi-bin/customer_service/complain.pl", but not to the script "/cgi-private/site_admin.pl". By default, path is set to "/", which causes the cookie to be sent to any CGI script on your site.

4. a "secure" flag

If the "secure" attribute is set, the cookie will only be sent to your script if the CGI request is occurring on a secure channel, such as SSL.

The interface to HTTP cookies is the `cookie()` method:

```
$cookie = $query->cookie(-name=>'sessionID',
                        -value=>'xyzy',
                        -expires=>'+1h',
                        -path=>'/cgi-bin/database',
                        -domain=>'.capricorn.org',
                        -secure=>1);
print $query->header(-cookie=>$cookie);
```

cookie() creates a new cookie. Its parameters include:

-name

The name of the cookie (required). This can be any string at all. Although browsers limit their cookie names to non-whitespace alphanumeric characters, CGI.pm removes this restriction by escaping and unescaping cookies behind the scenes.

-value

The value of the cookie. This can be any scalar value, array reference, or even associative array reference. For example, you can store an entire associative array into a cookie this way:

```
$cookie=$query->cookie(-name=>'family information',
                      -value=>\%childrens_ages);
```

-path

The optional partial path for which this cookie will be valid, as described above.

-domain

The optional partial domain for which this cookie will be valid, as described above.

-expires

The optional expiration date for this cookie. The format is as described in the section on the **header()** method:

```
"+1h" one hour from now
```

-secure

If set to true, this cookie will only be used within a secure SSL session.

The cookie created by **cookie()** must be incorporated into the HTTP header within the string returned by the **header()** method:

```
print $query->header(-cookie=>$my_cookie);
```

To create multiple cookies, give **header()** an array reference:

```
$cookie1 = $query->cookie(-name=>'riddle_name',
                        -value=>"The Sphynx's Question");
$cookie2 = $query->cookie(-name=>'answers',
                        -value=>\%answers);
print $query->header(-cookie=>[$cookie1,$cookie2]);
```

To retrieve a cookie, request it by name by calling **cookie()** method without the **-value** parameter:

```
use CGI;
$query = new CGI;
$riddle = $query->cookie('riddle_name');
%answers = $query->cookie('answers');
```

Cookies created with a single scalar value, such as the "riddle_name" cookie, will be returned in that form. Cookies with array and hash values can also be retrieved.

The cookie and CGI namespaces are separate. If you have a parameter named 'answers' and a cookie named 'answers', the values retrieved by **param()** and **cookie()** are independent of each other. However, it's

simple to turn a CGI parameter into a cookie, and vice-versa:

```
# turn a CGI parameter into a cookie
$c=$q->cookie(-name=>'answers',-value=>[$q->param('answers')]);
# vice-versa
$q->param(-name=>'answers',-value=>[$q->cookie('answers')]);
```

See the **cookie.cgi** example script for some ideas on how to use cookies effectively.

WORKING WITH FRAMES

It's possible for CGI.pm scripts to write into several browser panels and windows using the HTML 4 frame mechanism. There are three techniques for defining new frames programmatically:

1. Create a <Frameset document

After writing out the HTTP header, instead of creating a standard HTML document using the `start_html()` call, create a <frameset document that defines the frames on the page. Specify your script(s) (with appropriate parameters) as the SRC for each of the frames.

There is no specific support for creating <frameset sections in CGI.pm, but the HTML is very simple to write. See the frame documentation in Netscape's home pages for details

```
http://home.netscape.com/assist/net_sites/frames.html
```

2. Specify the destination for the document in the HTTP header

You may provide a **-target** parameter to the `header()` method:

```
print $q->header(-target=>'ResultsWindow');
```

This will tell the browser to load the output of your script into the frame named "ResultsWindow". If a frame of that name doesn't already exist, the browser will pop up a new window and load your script's document into that. There are a number of magic names that you can use for targets. See the frame documents on Netscape's home pages for details.

3. Specify the destination for the document in the <form tag

You can specify the frame to load in the FORM tag itself. With CGI.pm it looks like this:

```
print $q->start_form(-target=>'ResultsWindow');
```

When your script is reinvoked by the form, its output will be loaded into the frame named "ResultsWindow". If one doesn't already exist a new window will be created.

The script "frameset.cgi" in the examples directory shows one way to create pages in which the fill-out form and the response live in side-by-side frames.

LIMITED SUPPORT FOR CASCADING STYLE SHEETS

CGI.pm has limited support for HTML3's cascading style sheets (css). To incorporate a stylesheet into your document, pass the `start_html()` method a **-style** parameter. The value of this parameter may be a scalar, in which case it is incorporated directly into a <style section, or it may be a hash reference. In the latter case you should provide the hash with one or more of **-src** or **-code**. **-src** points to a URL where an externally-defined stylesheet can be found. **-code** points to a scalar value to be incorporated into a <style section. Style definitions in **-code** override similarly-named ones in **-src**, hence the name "cascading."

You may also specify the type of the stylesheet by adding the optional **-type** parameter to the hash pointed to by **-style**. If not specified, the style defaults to 'text/css'.

To refer to a style within the body of your document, add the **-class** parameter to any HTML element:

```
print h1({-class=>'Fancy'}, 'Welcome to the Party');
```

Or define styles on the fly with the **-style** parameter:

```
print h1({-style=>'Color: red;'}, 'Welcome to Hell');
```

You may also use the new **span()** element to apply a style to a section of text:

```
print span({-style=>'Color: red;'},
          h1('Welcome to Hell'),
          "Where did that handbasket get to?"
        );
```

Note that you must import the `":html3"` definitions to have the **span()** method available. Here's a quick and dirty example of using CSS's. See the CSS specification at <http://www.w3.org/pub/WWW/TR/Wd-css-1.html> for more information.

```
use CGI qw/:standard :html3/;

#here's a stylesheet incorporated directly into the page
$newStyle=<<END;
<!--
P.Tip {
    margin-right: 50pt;
    margin-left: 50pt;
    color: red;
}
P.Alert {
    font-size: 30pt;
    font-family: sans-serif;
    color: red;
}
-->
END
print header();
print start_html( -title=>'CGI with Style',
                  -style=>{-src=>'http://www.capricorn.com/style/st1.css',
                          -code=>$newStyle}
                );
print h1('CGI with Style'),
      p({-class=>'Tip'},
        "Better read the cascading style sheet spec before playing with this!"),
      span({-style=>'color: magenta'},
          "Look Mom, no hands!",
          p(),
          "Whooo wee!"
        );
print end_html;
```

Pass an array reference to **-style** in order to incorporate multiple stylesheets into your document.

DEBUGGING

If you are running the script from the command line or in the perl debugger, you can pass the script a list of keywords or parameter=value pairs on the command line or from standard input (you don't have to worry about tricking your script into reading from environment variables). You can pass keywords like this:

```
your_script.pl keyword1 keyword2 keyword3
```

or this:

```
your_script.pl keyword1+keyword2+keyword3
```

or this:

```
your_script.pl name1=value1 name2=value2
```

or this:

```
your_script.pl name1=value1&name2=value2
```

To turn off this feature, use the `--no_debug` pragma.

To test the POST method, you may enable full debugging with the `--debug` pragma. This will allow you to feed newline-delimited `name=value` pairs to the script on standard input.

When debugging, you can use quotes and backslashes to escape characters in the familiar shell manner, letting you place spaces and other funny characters in your `parameter=value` pairs:

```
your_script.pl "name1='I am a long value'" "name2=two\ words"
```

DUMPING OUT ALL THE NAME/VALUE PAIRS

The `Dump()` method produces a string consisting of all the query's name/value pairs formatted nicely as a nested list. This is useful for debugging purposes:

```
print $query->Dump
```

Produces something that looks like:

```
<ul>
<li>name1
  <ul>
    <li>value1
    <li>value2
  </ul>
<li>name2
  <ul>
    <li>value1
  </ul>
</ul>
```

As a shortcut, you can interpolate the entire CGI object into a string and it will be replaced with the a nice HTML dump shown above:

```
$query=new CGI;
print "<h2>Current Values</h2> $query\n";
```

FETCHING ENVIRONMENT VARIABLES

Some of the more useful environment variables can be fetched through this interface. The methods are as follows:

Accept()

Return a list of MIME types that the remote browser accepts. If you give this method a single argument corresponding to a MIME type, as in `$query->Accept('text/html')`, it will return a floating point value corresponding to the browser's preference for this type from 0.0 (don't want) to 1.0. Glob types (e.g. `text/*`) in the browser's accept list are handled correctly.

Note that the capitalization changed between version 2.43 and 2.44 in order to avoid conflict with Perl's `accept()` function.

raw_cookie()

Returns the `HTTP_COOKIE` variable, an HTTP extension implemented by Netscape browsers version 1.1 and higher, and all versions of Internet Explorer. Cookies have a special format, and this method call just returns the raw form (?cookie dough). See `cookie()` for ways of setting and retrieving cooked cookies.

Called with no parameters, `raw_cookie()` returns the packed cookie structure. You can separate it into individual cookies by splitting on the character sequence `;`. Called with the name of a cookie, retrieves the **unescaped** form of the cookie. You can use the regular `cookie()` method to get the

names, or use the `raw_fetch()` method from the `CGI::Cookie` module.

user_agent()

Returns the `HTTP_USER_AGENT` variable. If you give this method a single argument, it will attempt to pattern match on it, allowing you to do something like `$query-user_agent(netscape);`

path_info()

Returns additional path information from the script URL. E.G. fetching `/cgi-bin/your_script/additional/stuff` will result in `$query-path_info()` returning `"/additional/stuff"`.

NOTE: The Microsoft Internet Information Server is broken with respect to additional path information. If you use the Perl DLL library, the IIS server will attempt to execute the additional path information as a Perl script. If you use the ordinary file associations mapping, the path information will be present in the environment, but incorrect. The best thing to do is to avoid using additional path information in CGI scripts destined for use with IIS.

path_translated()

As per `path_info()` but returns the additional path information translated into a physical path, e.g. `"/usr/local/etc/httpd/htdocs/additional/stuff"`.

The Microsoft IIS is broken with respect to the translated path as well.

remote_host()

Returns either the remote host name or IP address, if the former is unavailable.

script_name()

Return the script name as a partial URL, for self-referring scripts.

referer()

Return the URL of the page the browser was viewing prior to fetching your script. Not available for all browsers.

auth_type()

Return the authorization/verification method in use for this script, if any.

server_name()

Returns the name of the server, usually the machine's host name.

virtual_host()

When using virtual hosts, returns the name of the host that the browser attempted to contact

server_port()

Return the port that the server is listening on.

server_software()

Returns the server software and version number.

remote_user()

Return the authorization/verification name used for user verification, if this script is protected.

user_name()

Attempt to obtain the remote user's name, using a variety of different techniques. This only works with older browsers such as Mosaic. Newer browsers do not report the user name for privacy reasons!

request_method()

Returns the method used to access your script, usually one of 'POST', 'GET' or 'HEAD'.

content_type()

Returns the content_type of data submitted in a POST, generally `multipart/form-data` or `application/x-www-form-urlencoded`

http()

Called with no arguments returns the list of HTTP environment variables, including such things as `HTTP_USER_AGENT`, `HTTP_ACCEPT_LANGUAGE`, and `HTTP_ACCEPT_CHARSET`, corresponding to the like-named HTTP header fields in the request. Called with the name of an HTTP header field, returns its value. Capitalization and the use of hyphens versus underscores are not significant.

For example, all three of these examples are equivalent:

```
$requested_language = $q->http('Accept-language');
$requested_language = $q->http('Accept_language');
$requested_language = $q->http('HTTP_ACCEPT_LANGUAGE');
```

https()

The same as `http()`, but operates on the HTTPS environment variables present when the SSL protocol is in effect. Can be used to determine whether SSL is turned on.

USING NPH SCRIPTS

NPH, or "no-parsed-header", scripts bypass the server completely by sending the complete HTTP header directly to the browser. This has slight performance benefits, but is of most use for taking advantage of HTTP extensions that are not directly supported by your server, such as server push and PICS headers.

Servers use a variety of conventions for designating CGI scripts as NPH. Many Unix servers look at the beginning of the script's name for the prefix "nph-". The Macintosh WebSTAR server and Microsoft's Internet Information Server, in contrast, try to decide whether a program is an NPH script by examining the first line of script output.

CGI.pm supports NPH scripts with a special NPH mode. When in this mode, CGI.pm will output the necessary extra header information when the `header()` and `redirect()` methods are called.

The Microsoft Internet Information Server requires NPH mode. As of version 2.30, CGI.pm will automatically detect when the script is running under IIS and put itself into this mode. You do not need to do this manually, although it won't hurt anything if you do. However, note that if you have applied Service Pack 6, much of the functionality of NPH scripts, including the ability to redirect while setting a cookie, **do not work at all on IIS without a special patch from Microsoft.** See

<http://support.microsoft.com/support/kb/articles/Q280/3/41.ASP>: Non-Parsed Headers Stripped From CGI Applications That Have nph- Prefix in Name.

In the `use` statement

Simply add the "nph" pragma to the list of symbols to be imported into your script:

```
use CGI qw(:standard -nph)
```

By calling the `nph()` method:

Call `nph()` with a non-zero parameter at any point after using CGI.pm in your program.

```
CGI->nph(1)
```

By using `-nph` parameters

in the `header()` and `redirect()` statements:

```
print $q->header(-nph=>1);
```

Server Push

CGI.pm provides four simple functions for producing multipart documents of the type needed to implement server push. These functions were graciously provided by Ed Jordan <ed@fidalgo.net>. To import these into your namespace, you must import the "push" set. You are also advised to put the script into NPH mode and to set `$|` to 1 to avoid buffering problems.

Here is a simple script that demonstrates server push:

```
#!/usr/local/bin/perl
use CGI qw/:push -nph/;
$| = 1;
print multipart_init(-boundary=>'----here we go!');
foreach (0 .. 4) {
    print multipart_start(-type=>'text/plain'),
        "The current time is ", scalar(localtime), "\n";
    if ($_ < 4) {
        print multipart_end;
    } else {
        print multipart_final;
    }
    sleep 1;
}
```

This script initializes server push by calling `multipart_init()`. It then enters a loop in which it begins a new multipart section by calling `multipart_start()`, prints the current local time, and ends a multipart section with `multipart_end()`. It then sleeps a second, and begins again. On the final iteration, it ends the multipart section with `multipart_final()` rather than with `multipart_end()`.

```
multipart_init()
    multipart_init(-boundary=>$boundary);
```

Initialize the multipart system. The `-boundary` argument specifies what MIME boundary string to use to separate parts of the document. If not provided, CGI.pm chooses a reasonable boundary for you.

```
multipart_start()
    multipart_start(-type=>$type)
```

Start a new part of the multipart document using the specified MIME type. If not specified, text/html is assumed.

```
multipart_end()
    multipart_end()
```

End a part. You must remember to call `multipart_end()` once for each `multipart_start()`, except at the end of the last part of the multipart document when `multipart_final()` should be called instead of `multipart_end()`.

```
multipart_final()
    multipart_final()
```

End all parts. You should call `multipart_final()` rather than `multipart_end()` at the end of the last part of the multipart document.

Users interested in server push applications should also have a look at the CGI::Push module.

Only Netscape Navigator supports server push. Internet Explorer browsers do not.

Avoiding Denial of Service Attacks

A potential problem with CGI.pm is that, by default, it attempts to process form POSTings no matter how large they are. A wily hacker could attack your site by sending a CGI script a huge POST of many megabytes. CGI.pm will attempt to read the entire POST into a variable, growing hugely in size until it runs out of memory. While the script attempts to allocate the memory the system may slow down dramatically. This is a form of denial of service attack.

Another possible attack is for the remote user to force CGI.pm to accept a huge file upload. CGI.pm will accept the upload and store it in a temporary directory even if your script doesn't expect to receive an uploaded file. CGI.pm will delete the file automatically when it terminates, but in the meantime the remote user may have filled up the server's disk space, causing problems for other programs.

The best way to avoid denial of service attacks is to limit the amount of memory, CPU time and disk space that CGI scripts can use. Some Web servers come with built-in facilities to accomplish this. In other cases, you can use the shell *limit* or *ulimit* commands to put ceilings on CGI resource usage.

CGI.pm also has some simple built-in protections against denial of service attacks, but you must activate them before you can use them. These take the form of two global variables in the CGI name space:

\$CGI::POST_MAX

If set to a non-negative integer, this variable puts a ceiling on the size of POSTings, in bytes. If CGI.pm detects a POST that is greater than the ceiling, it will immediately exit with an error message. This value will affect both ordinary POSTs and multipart POSTs, meaning that it limits the maximum size of file uploads as well. You should set this to a reasonably high value, such as 1 megabyte.

\$CGI::DISABLE_UPLOADS

If set to a non-zero value, this will disable file uploads completely. Other fill-out form values will work as usual.

You can use these variables in either of two ways.

1. On a script-by-script basis

Set the variable at the top of the script, right after the "use" statement:

```
use CGI qw/:standard/;
use CGI::Carp 'fatalToBrowser';
$CGI::POST_MAX=1024 * 100; # max 100K posts
$CGI::DISABLE_UPLOADS = 1; # no uploads
```

2. Globally for all scripts

Open up CGI.pm, find the definitions for \$POST_MAX and \$DISABLE_UPLOADS, and set them to the desired values. You'll find them towards the top of the file in a subroutine named `initialize_globals()`.

An attempt to send a POST larger than \$POST_MAX bytes will cause `param()` to return an empty CGI parameter list. You can test for this event by checking `cgi_error()`, either after you create the CGI object or, if you are using the function-oriented interface, call `<param()` for the first time. If the POST was intercepted, then `cgi_error()` will return the message "413 POST too large".

This error message is actually defined by the HTTP protocol, and is designed to be returned to the browser as the CGI script's status code. For example:

```
$uploaded_file = param('upload');
if (!$uploaded_file && cgi_error()) {
    print header(-status=>cgi_error());
    exit 0;
}
```

However it isn't clear that any browser currently knows what to do with this status code. It might be better just to create an HTML page that warns the user of the problem.

COMPATIBILITY WITH CGI-LIB.PL

To make it easier to port existing programs that use `cgi-lib.pl` the compatibility routine "ReadParse" is provided. Porting is simple:

OLD VERSION

```
require "cgi-lib.pl";
&ReadParse;
print "The value of the antique is $in{antique}.\n";
```

NEW VERSION

```
use CGI;
```

```
CGI::ReadParse
print "The value of the antique is $in{antique}.\n";
```

CGI.pm's `ReadParse()` routine creates a tied variable named `%in`, which can be accessed to obtain the query variables. Like `ReadParse`, you can also provide your own variable. Infrequently used features of `ReadParse`, such as the creation of `@in` and `$in` variables, are not supported.

Once you use `ReadParse`, you can retrieve the query object itself this way:

```
$q = $in{CGI};
print $q->textfield(-name=>'wow',
                  -value=>'does this really work?');
```

This allows you to start using the more interesting features of CGI.pm without rewriting your old scripts from scratch.

AUTHOR INFORMATION

Copyright 1995–1998, Lincoln D. Stein. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Address bug reports and comments to: lstein@cshl.org. When sending bug reports, please provide the version of CGI.pm, the version of Perl, the name and version of your Web server, and the name and version of the operating system you are using. If the problem is even remotely browser dependent, please provide information about the affected browsers as well.

CREDITS

Thanks very much to:

Matt Heffron (heffron@falstaff.css.beckman.com)
 James Taylor (james.taylor@srs.gov)
 Scott Anguish (sanguish@digifix.com)
 Mike Jewell (mlj3u@virginia.edu)
 Timothy Shimmin (tes@kbs.citri.edu.au)
 Joergen Haegg (jh@axis.se)
 Laurent Delfosse (delfosse@delfosse.com)
 Richard Resnick (applepi1@aol.com)
 Craig Bishop (csb@barwonwater.vic.gov.au)
 Tony Curtis (tc@vcpc.univie.ac.at)
 Tim Bunce (Tim.Bunce@ig.co.uk)
 Tom Christiansen (tchrist@convex.com)
 Andreas Koenig (k@franz.ww.TU-Berlin.DE)
 Tim MacKenzie (Tim.MacKenzie@fulcrum.com.au)
 Kevin B. Hendricks (kbhend@dogwood.tyler.wm.edu)
 Stephen Dahmen (joyfire@inxpress.net)
 Ed Jordan (ed@fidalgo.net)
 David Alan Pisoni (david@cnation.com)
 Doug MacEachern (doug@opengroup.org)
 Robin Houston (robin@oneworld.org)
 ...and many many more...

for suggestions and bug fixes.

A COMPLETE EXAMPLE OF A SIMPLE FORM-BASED SCRIPT

```
#!/usr/local/bin/perl

use CGI;

$query = new CGI;

print $query->header;
```

```

print $query->start_html("Example CGI.pm Form");
print "<h1> Example CGI.pm Form</h1>\n";
&print_prompt($query);
&do_work($query);
&print_tail;
print $query->end_html;

sub print_prompt {
    my($query) = @_;

    print $query->start_form;
    print "<em>What's your name?</em><br>";
    print $query->textfield('name');
    print $query->checkbox('Not my real name');

    print "<p><em>Where can you find English Sparrows?</em><br>";
    print $query->checkbox_group(
        -name=>'Sparrow locations',
        -values=>[England,France,Spain,Asia,Hoboken],
        -linebreak=>'yes',
        -defaults=>[England,Asia]);

    print "<p><em>How far can they fly?</em><br>",
        $query->radio_group(
            -name=>'how far',
            -values=>['10 ft','1 mile','10 miles','real far'],
            -default=>'1 mile');

    print "<p><em>What's your favorite color?</em> ";
    print $query->popup_menu(-name=>'Color',
        -values=>['black','brown','red','yellow'],
        -default=>'red');

    print $query->hidden('Reference','Monty Python and the Holy Grail');

    print "<p><em>What have you got there?</em><br>";
    print $query->scrolling_list(
        -name=>'possessions',
        -values=>['A Coconut','A Grail','An Icon',
            'A Sword','A Ticket'],
        -size=>5,
        -multiple=>'true');

    print "<p><em>Any parting comments?</em><br>";
    print $query->textarea(-name=>'Comments',
        -rows=>10,
        -columns=>50);

    print "<p>",$query->reset;
    print $query->submit('Action','Shout');
    print $query->submit('Action','Scream');
    print $query->endform;
    print "<hr>\n";
}

sub do_work {
    my($query) = @_;
    my(@values,$key);

```

```
print "<h2>Here are the current settings in this form</h2>";

foreach $key ($query->param) {
    print "<strong>$key</strong> -> ";
    @values = $query->param($key);
    print join(", ", @values), "<br>\n";
}

sub print_tail {
    print <<END;
    <hr>
    <address>Lincoln D. Stein</address><br>
    <a href="/">Home Page</a>
    END
}
```

BUGS

This module has grown large and monolithic. Furthermore it's doing many things, such as handling URLs, parsing CGI input, writing HTML, etc., that are also done in the LWP modules. It should be discarded in favor of the CGI::* modules, but somehow I continue to work on it.

Note that the code is truly contorted in order to avoid spurious warnings when programs are run with the `-w` switch.

SEE ALSO

[CGI::Carp](#), [CGI::Fast](#), [CGI::Pretty](#)

NAME

CGI::Apache – Backward compatibility module for CGI.pm

SYNOPSIS

Do not use this module. It is deprecated.

ABSTRACT**DESCRIPTION****AUTHOR INFORMATION****BUGS****SEE ALSO**

NAME

CGI::Carp – CGI routines for writing to the HTTPD (or other) error log

SYNOPSIS

```
use CGI::Carp;

croak "We're outta here!";
confess "It was my fault: $!";
carp "It was your fault!";
warn "I'm confused";
die "I'm dying.\n";

use CGI::Carp qw(cluck);
cluck "I wouldn't do that if I were you";

use CGI::Carp qw(fatalsToBrowser);
die "Fatal error messages are now sent to browser";
```

DESCRIPTION

CGI scripts have a nasty habit of leaving warning messages in the error logs that are neither time stamped nor fully identified. Tracking down the script that caused the error is a pain. This fixes that. Replace the usual

```
use Carp;
```

with

```
use CGI::Carp
```

And the standard `warn()`, `die()`, `croak()`, `confess()` and `carp()` calls will automatically be replaced with functions that write out nicely time-stamped messages to the HTTP server error log.

For example:

```
[Fri Nov 17 21:40:43 1995] test.pl: I'm confused at test.pl line 3.
[Fri Nov 17 21:40:43 1995] test.pl: Got an error message: Permission denied.
[Fri Nov 17 21:40:43 1995] test.pl: I'm dying.
```

REDIRECTING ERROR MESSAGES

By default, error messages are sent to `STDERR`. Most HTTPD servers direct `STDERR` to the server's error log. Some applications may wish to keep private error logs, distinct from the server's error log, or they may wish to direct error messages to `STDOUT` so that the browser will receive them.

The `carpout()` function is provided for this purpose. Since `carpout()` is not exported by default, you must import it explicitly by saying

```
use CGI::Carp qw(carpout);
```

The `carpout()` function requires one argument, which should be a reference to an open filehandle for writing errors. It should be called in a `BEGIN` block at the top of the CGI application so that compiler errors will be caught. Example:

```
BEGIN {
    use CGI::Carp qw(carpout);
    open(LOG, ">>/usr/local/cgi-logs/mycgi-log") or
        die("Unable to open mycgi-log: $!\n");
    carpout(LOG);
}
```

`carpout()` does not handle file locking on the log for you at this point.

The real `STDERR` is not closed — it is moved to `CGI::Carp::SAVEERR`. Some servers, when dealing with

CGI scripts, close their connection to the browser when the script closes STDOUT and STDERR. CGI::Carp::SAVEERR is there to prevent this from happening prematurely.

You can pass filehandles to `carpout()` in a variety of ways. The "correct" way according to Tom Christiansen is to pass a reference to a filehandle GLOB:

```
carpout(\*LOG);
```

This looks weird to mere mortals however, so the following syntaxes are accepted as well:

```
carpout(LOG);
carpout(main::LOG);
carpout(main'LOG);
carpout(\LOG);
carpout(\'main::LOG');
... and so on
```

FileHandle and other objects work as well.

Use of `carpout()` is not great for performance, so it is recommended for debugging purposes or for moderate-use applications. A future version of this module may delay redirecting STDERR until one of the CGI::Carp methods is called to prevent the performance hit.

MAKING PERL ERRORS APPEAR IN THE BROWSER WINDOW

If you want to send fatal (die, confess) errors to the browser, ask to import the special "fatalsToBrowser" subroutine:

```
use CGI::Carp qw(fatalsToBrowser);
die "Bad error here";
```

Fatal errors will now be echoed to the browser as well as to the log. CGI::Carp arranges to send a minimal HTTP header to the browser so that even errors that occur in the early compile phase will be seen. Nonfatal errors will still be directed to the log file only (unless redirected with `carpout`).

Changing the default message

By default, the software error message is followed by a note to contact the Webmaster by e-mail with the time and date of the error. If this message is not to your liking, you can change it using the `set_message()` routine. This is not imported by default; you should import it on the `use()` line:

```
use CGI::Carp qw(fatalsToBrowser set_message);
set_message("It's not a bug, it's a feature!");
```

You may also pass in a code reference in order to create a custom error message. At run time, your code will be called with the text of the error message that caused the script to die. Example:

```
use CGI::Carp qw(fatalsToBrowser set_message);
BEGIN {
    sub handle_errors {
        my $msg = shift;
        print "<h1>Oh gosh</h1>";
        print "<p>Got an error: $msg</p>";
    }
    set_message(\&handle_errors);
}
```

In order to correctly intercept compile-time errors, you should call `set_message()` from within a `BEGIN{}` block.

MAKING WARNINGS APPEAR AS HTML COMMENTS

It is now also possible to make non-fatal errors appear as HTML comments embedded in the output of your program. To enable this feature, export the new "warningsToBrowser" subroutine. Since sending warnings to the browser before the HTTP headers have been sent would cause an error, any warnings are stored in an internal buffer until you call the `warningsToBrowser()` subroutine with a true argument:

```
use CGI::Carp qw(fatalsToBrowser warningsToBrowser);
use CGI qw(:standard);
print header();
warningsToBrowser(1);
```

You may also give a false argument to `warningsToBrowser()` to prevent warnings from being sent to the browser while you are printing some content where HTML comments are not allowed:

```
warningsToBrowser(0);    # disable warnings
print "<script type=\"text/javascript\"><!--\n";
print_some_javascript_code();
print "//--></script>\n";
warningsToBrowser(1);    # re-enable warnings
```

Note: In this respect `warningsToBrowser()` differs fundamentally from `fatalsToBrowser()`, which you should never call yourself!

CHANGE LOG

- 1.05 `carpout()` added and minor corrections by Marc Hedlund
<hedlund@best.com on 11/26/95.
- 1.06 `fatalsToBrowser()` no longer aborts for fatal errors within
`eval()` statements.
- 1.08 `set_message()` added and `carpout()` expanded to allow for `FileHandle`
objects.
- 1.09 `set_message()` now allows users to pass a code REFERENCE for
really custom error messages. `croak` and `carp` are now
exported by default. Thanks to Gunther Birznies for the
patches.
- 1.10 Patch from Chris Dean (ctdean@cogit.com) to allow
module to run correctly under `mod_perl`.
- 1.11 Changed order of `>` and `<` escapes.
- 1.12 Changed `die()` on line 217 to `CORE::die` to avoid `-w` warning.
- 1.13 Added `cluck()` to make the module orthogonal with `Carp`.
More `mod_perl` related fixes.
- 1.20 Patch from Ilmari Karonen (perl@itz.pp.sci.fi): Added
`warningsToBrowser()`. Replaced `<CODE` tags with `<PRE` in
`fatalsToBrowser()` output.
- 1.23 `ineval()` now checks both `$_$` and inspects the message for the "eval" pattern
(hack alert!) in order to accomodate various combinations of Perl and
`mod_perl`.

AUTHORS

Copyright 1995–2002, Lincoln D. Stein. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Address bug reports and comments to: lstein@cshl.org

SEE ALSO

Carp, CGI::Base, CGI::BasePlus, CGI::Request, CGI::MiniSvr, CGI::Form, CGI::Response

NAME

CGI::Cookie – Interface to Netscape Cookies

SYNOPSIS

```
use CGI qw/:standard/;
use CGI::Cookie;

# Create new cookies and send them
$cookie1 = new CGI::Cookie(-name=>'ID',-value=>123456);
$cookie2 = new CGI::Cookie(-name=>'preferences',
                           -value=>{ font => Helvetica,
                                       size => 12 }
                           );
print header(-cookie=>[$cookie1,$cookie2]);

# fetch existing cookies
%cookies = fetch CGI::Cookie;
$id = $cookies{'ID'}->value;

# create cookies returned from an external source
%cookies = parse CGI::Cookie($ENV{COOKIE});
```

DESCRIPTION

CGI::Cookie is an interface to Netscape (HTTP/1.1) cookies, an innovation that allows Web servers to store persistent information on the browser's side of the connection. Although CGI::Cookie is intended to be used in conjunction with CGI.pm (and is in fact used by it internally), you can use this module independently.

For full information on cookies see

<http://www.ics.uci.edu/pub/ietf/http/rfc2109.txt>

USING CGI::Cookie

CGI::Cookie is object oriented. Each cookie object has a name and a value. The name is any scalar value. The value is any scalar or array value (associative arrays are also allowed). Cookies also have several optional attributes, including:

1. expiration date

The expiration date tells the browser how long to hang on to the cookie. If the cookie specifies an expiration date in the future, the browser will store the cookie information in a disk file and return it to the server every time the user reconnects (until the expiration date is reached). If the cookie specifies an expiration date in the past, the browser will remove the cookie from the disk file. If the expiration date is not specified, the cookie will persist only until the user quits the browser.

2. domain

This is a partial or complete domain name for which the cookie is valid. The browser will return the cookie to any host that matches the partial domain name. For example, if you specify a domain name of ".capricorn.com", then Netscape will return the cookie to Web servers running on any of the machines "www.capricorn.com", "ftp.capricorn.com", "feckless.capricorn.com", etc. Domain names must contain at least two periods to prevent attempts to match on top level domains like ".edu". If no domain is specified, then the browser will only return the cookie to servers on the host the cookie originated from.

3. path

If you provide a cookie path attribute, the browser will check it against your script's URL before returning the cookie. For example, if you specify the path "/cgi-bin", then the cookie will be returned to each of the scripts "/cgi-bin/tally.pl", "/cgi-bin/order.pl", and "/cgi-bin/customer_service/complain.pl", but not to the script "/cgi-private/site_admin.pl". By default, the path is set to "/", so that all scripts at your site will receive the cookie.

4. secure flag

If the "secure" attribute is set, the cookie will only be sent to your script if the CGI request is occurring on a secure channel, such as SSL.

Creating New Cookies

```
$c = new CGI::Cookie(-name    => 'foo',
                    -value    => 'bar',
                    -expires  => '+3M',
                    -domain   => '.capricorn.com',
                    -path     => '/cgi-bin/database',
                    -secure   => 1
                    );
```

Create cookies from scratch with the **new** method. The **-name** and **-value** parameters are required. The name must be a scalar value. The value can be a scalar, an array reference, or a hash reference. (At some point in the future cookies will support one of the Perl object serialization protocols for full generality).

-expires accepts any of the relative or absolute date formats recognized by CGI.pm, for example "+3M" for three months in the future. See CGI.pm's documentation for details.

-domain points to a domain name or to a fully qualified host name. If not specified, the cookie will be returned only to the Web server that created it.

-path points to a partial URL on the current server. The cookie will be returned to all URLs beginning with the specified path. If not specified, it defaults to '/', which returns the cookie to all pages at your site.

-secure if set to a true value instructs the browser to return the cookie only when a cryptographic protocol is in use.

Sending the Cookie to the Browser

Within a CGI script you can send a cookie to the browser by creating one or more Set-Cookie: fields in the HTTP header. Here is a typical sequence:

```
my $c = new CGI::Cookie(-name    => 'foo',
                      -value    => ['bar', 'baz'],
                      -expires  => '+3M');

print "Set-Cookie: $c\n";
print "Content-Type: text/html\n\n";
```

To send more than one cookie, create several Set-Cookie: fields. Alternatively, you may concatenate the cookies together with ";" and send them in one field.

If you are using CGI.pm, you send cookies by providing a **-cookie** argument to the **header()** method:

```
print header(-cookie=>$c);
```

Mod_perl users can set cookies using the request object's **header_out()** method:

```
$r->header_out('Set-Cookie', $c);
```

Internally, Cookie overloads the "" operator to call its **as_string()** method when incorporated into the HTTP header. **as_string()** turns the Cookie's internal representation into an RFC-compliant text representation. You may call **as_string()** yourself if you prefer:

```
print "Set-Cookie: ", $c->as_string, "\n";
```

Recovering Previous Cookies

```
%cookies = fetch CGI::Cookie;
```

fetch returns an associative array consisting of all cookies returned by the browser. The keys of the array are the cookie names. You can iterate through the cookies this way:

```
%cookies = fetch CGI::Cookie;
foreach (keys %cookies) {
    do_something($cookies{$_});
}
```

In a scalar context, `fetch()` returns a hash reference, which may be more efficient if you are manipulating multiple cookies.

CGI.pm uses the URL escaping methods to save and restore reserved characters in its cookies. If you are trying to retrieve a cookie set by a foreign server, this escaping method may trip you up. Use `raw_fetch()` instead, which has the same semantics as `fetch()`, but performs no unescaping.

You may also retrieve cookies that were stored in some external form using the `parse()` class method:

```
$COOKIES = `cat /usr/tmp/Cookie_stash`;
%cookies = parse CGI::Cookie($COOKIES);
```

Manipulating Cookies

Cookie objects have a series of accessor methods to get and set cookie attributes. Each accessor has a similar syntax. Called without arguments, the accessor returns the current value of the attribute. Called with an argument, the accessor changes the attribute and returns its new value.

name()

Get or set the cookie's name. Example:

```
$name = $c->name;
$new_name = $c->name('fred');
```

value()

Get or set the cookie's value. Example:

```
$value = $c->value;
@new_value = $c->value(['a','b','c','d']);
```

value() is context sensitive. In a list context it will return the current value of the cookie as an array. In a scalar context it will return the **first** value of a multivalued cookie.

domain()

Get or set the cookie's domain.

path()

Get or set the cookie's path.

expires()

Get or set the cookie's expiration time.

AUTHOR INFORMATION

Copyright 1997–1998, Lincoln D. Stein. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Address bug reports and comments to: lstein@cshl.org

BUGS

This section intentionally left blank.

SEE ALSO

CGI::Carp, *CGI*

NAME

CGI::Fast – CGI Interface for Fast CGI

SYNOPSIS

```
use CGI::Fast qw(:standard);
$COUNTER = 0;
while (new CGI::Fast) {
    print header;
    print start_html("Fast CGI Rocks");
    print
        h1("Fast CGI Rocks"),
        "Invocation number ",b($COUNTER++),
        " PID ",b($$), ". ",
        hr;
    print end_html;
}
```

DESCRIPTION

CGI::Fast is a subclass of the CGI object created by CGI.pm. It is specialized to work well with the Open Market FastCGI standard, which greatly speeds up CGI scripts by turning them into persistently running server processes. Scripts that perform time-consuming initialization processes, such as loading large modules or opening persistent database connections, will see large performance improvements.

OTHER PIECES OF THE PUZZLE

In order to use CGI::Fast you'll need a FastCGI-enabled Web server. Open Market's server is FastCGI-savvy. There are also freely redistributable FastCGI modules for NCSA httpd 1.5 and Apache. FastCGI-enabling modules for Microsoft Internet Information Server and Netscape Communications Server have been announced.

In addition, you'll need a version of the Perl interpreter that has been linked with the FastCGI I/O library. Precompiled binaries are available for several platforms, including DEC Alpha, HP-UX and SPARC/Solaris, or you can rebuild Perl from source with patches provided in the FastCGI developer's kit. The FastCGI Perl interpreter can be used in place of your normal Perl without ill consequences.

You can find FastCGI modules for Apache and NCSA httpd, precompiled Perl interpreters, and the FastCGI developer's kit all at URL:

<http://www.fastcgi.com/>

WRITING FASTCGI PERL SCRIPTS

FastCGI scripts are persistent: one or more copies of the script are started up when the server initializes, and stay around until the server exits or they die a natural death. After performing whatever one-time initialization it needs, the script enters a loop waiting for incoming connections, processing the request, and waiting some more.

A typical FastCGI script will look like this:

```
#!/usr/local/bin/perl    # must be a FastCGI version of perl!
use CGI::Fast;
&do_some_initialization();
while ($q = new CGI::Fast) {
    &process_request($q);
}
```

Each time there's a new request, CGI::Fast returns a CGI object to your loop. The rest of the time your script waits in the call to new(). When the server requests that your script be terminated, new() will return undef. You can of course exit earlier if you choose. A new version of the script will be respawned to take its place (this may be necessary in order to avoid Perl memory leaks in long-running scripts).

CGI.pm's default CGI object mode also works. Just modify the loop this way:

```
while (new CGI::Fast) {
    &process_request;
}
```

Calls to `header()`, `start_form()`, etc. will all operate on the current request.

INSTALLING FASTCGI SCRIPTS

See the FastCGI developer's kit documentation for full details. On the Apache server, the following line must be added to `srn.conf`:

```
AddType application/x-httpd-fcgi .fcgi
```

FastCGI scripts must end in the extension `.fcgi`. For each script you install, you must add something like the following to `srn.conf`:

```
FastCgiServer /usr/etc/httpd/fcgi-bin/file_upload.fcgi -processes 2
```

This instructs Apache to launch two copies of `file_upload.fcgi` at startup time.

USING FASTCGI SCRIPTS AS CGI SCRIPTS

Any script that works correctly as a FastCGI script will also work correctly when installed as a vanilla CGI script. However it will not see any performance benefit.

EXTERNAL FASTCGI SERVER INVOCATION

FastCGI supports a TCP/IP transport mechanism which allows FastCGI scripts to run external to the webserver, perhaps on a remote machine. To configure the webserver to connect to an external FastCGI server, you would add the following to your `srn.conf`:

```
FastCgiExternalServer /usr/etc/httpd/fcgi-bin/file_upload.fcgi -host sputnik:8888
```

Two environment variables affect how the `CGI::Fast` object is created, allowing `CGI::Fast` to be used as an external FastCGI server. (See `FCGI` documentation for `FCGI::OpenSocket` for more information.)

FCGI_SOCKET_PATH

The address (TCP/IP) or path (UNIX Domain) of the socket the external FastCGI script to which bind can listen for incoming connections from the web server.

FCGI_LISTEN_QUEUE

Maximum length of the queue of pending connections.

For example:

```
#!/usr/local/bin/perl    # must be a FastCGI version of perl!
use CGI::Fast;
&do_some_initialization();
$ENV{FCGI_SOCKET_PATH} = "sputnik:8888";
$ENV{FCGI_LISTEN_QUEUE} = 100;
while ($q = new CGI::Fast) {
    &process_request($q);
}
```

CAVEATS

I haven't tested this very much.

AUTHOR INFORMATION

Copyright 1996–1998, Lincoln D. Stein. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Address bug reports and comments to: lstein@cshl.org

BUGS

This section intentionally left blank.

SEE ALSO

[CGI::Carp](#), *[CGI](#)*

NAME

CGI::Pretty – module to produce nicely formatted HTML code

SYNOPSIS

```
use CGI::Pretty qw( :html3 );

# Print a table with a single data element
print table( TR( td( "foo" ) ) );
```

DESCRIPTION

CGI::Pretty is a module that derives from CGI. It's sole function is to allow users of CGI to output nicely formatted HTML code.

When using the CGI module, the following code:

```
print table( TR( td( "foo" ) ) );
```

produces the following output:

```
<TABLE<TR<TDfoo</TD</TR</TABLE
```

If a user were to create a table consisting of many rows and many columns, the resultant HTML code would be quite difficult to read since it has no carriage returns or indentation.

CGI::Pretty fixes this problem. What it does is add a carriage return and indentation to the HTML code so that one can easily read it.

```
print table( TR( td( "foo" ) ) );
```

now produces the following output:

```
<TABLE
<TR
  <TD
    foo
  </TD
</TR
</TABLE
```

Tags that won't be formatted

The <A and <PRE tags are not formatted. If these tags were formatted, the user would see the extra indentation on the web browser causing the page to look different than what would be expected. If you wish to add more tags to the list of tags that are not to be touched, push them onto the @AS_IS array:

```
push @CGI::Pretty::AS_IS, qw( CODE XMP );
```

Customizing the Indenting

If you wish to have your own personal style of indenting, you can change the \$INDENT variable:

```
$CGI::Pretty::INDENT = "\t\t";
```

would cause the indents to be two tabs.

Similarly, if you wish to have more space between lines, you may change the \$LINEBREAK variable:

```
$CGI::Pretty::LINEBREAK = "\n\n";
```

would create two carriage returns between lines.

If you decide you want to use the regular CGI indenting, you can easily do the following:

```
$CGI::Pretty::INDENT = $CGI::Pretty::LINEBREAK = "";
```


BUGS

This section intentionally left blank.

AUTHOR

Brian Paulsen <Brian@ThePaulsens.com, with minor modifications by Lincoln Stein <lstein@cshl.org for incorporation into the CGI.pm distribution.

Copyright 1999, Brian Paulsen. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Bug reports and comments to Brian@ThePaulsens.com. You can also write to lstein@cshl.org, but this code looks pretty hairy to me and I'm not sure I understand it!

SEE ALSO

[CGI](#)

NAME

CGI::Push – Simple Interface to Server Push

SYNOPSIS

```
use CGI::Push qw(:standard);

do_push(-next_page=>\&next_page,
        -last_page=>\&last_page,
        -delay=>0.5);

sub next_page {
    my($q,$counter) = @_;
    return undef if $counter >= 10;
    return start_html('Test'),
           h1('Visible'),"\n",
           "This page has been called ", strong($counter)," times",
           end_html();
}

sub last_page {
    my($q,$counter) = @_;
    return start_html('Done'),
           h1('Finished'),
           strong($counter - 1),' iterations.',
           end_html;
}
```

DESCRIPTION

CGI::Push is a subclass of the CGI object created by CGI.pm. It is specialized for server push operations, which allow you to create animated pages whose content changes at regular intervals.

You provide CGI::Push with a pointer to a subroutine that will draw one page. Every time your subroutine is called, it generates a new page. The contents of the page will be transmitted to the browser in such a way that it will replace what was there beforehand. The technique will work with HTML pages as well as with graphics files, allowing you to create animated GIFs.

Only Netscape Navigator supports server push. Internet Explorer browsers do not.

USING CGI::Push

CGI::Push adds one new method to the standard CGI suite, `do_push()`. When you call this method, you pass it a reference to a subroutine that is responsible for drawing each new page, an interval delay, and an optional subroutine for drawing the last page. Other optional parameters include most of those recognized by the `CGI::header()` method.

You may call `do_push()` in the object oriented manner or not, as you prefer:

```
use CGI::Push;
$q = new CGI::Push;
$q->do_push(-next_page=>\&draw_a_page);

-or-

use CGI::Push qw(:standard);
do_push(-next_page=>\&draw_a_page);
```

Parameters are as follows:

```
-next_page
    do_push(-next_page=>\&my_draw_routine);
```

This required parameter points to a reference to a subroutine responsible for drawing each new page. The subroutine should expect two parameters consisting of the CGI object and a counter indicating the number of times the subroutine has been called. It should return the contents of the page as an **array** of one or more items to print. It can return a false value (or an empty array) in order to abort the redrawing loop and print out the final page (if any)

```
sub my_draw_routine {
    my($q,$counter) = @_;
    return undef if $counter > 100;
    return start_html('testing'),
           h1('testing'),
           "This page called $counter times";
}
```

You are of course free to refer to create and use global variables within your draw routine in order to achieve special effects.

-last_page

This optional parameter points to a reference to the subroutine responsible for drawing the last page of the series. It is called after the `-next_page` routine returns a false value. The subroutine itself should have exactly the same calling conventions as the `-next_page` routine.

-type

This optional parameter indicates the content type of each page. It defaults to "text/html". Normally the module assumes that each page is of a homogenous MIME type. However if you provide either of the magic values "heterogeneous" or "dynamic" (the latter provided for the convenience of those who hate long parameter names), you can specify the MIME type — and other header fields — on a per-page basis. See "heterogeneous pages" for more details.

-delay

This indicates the delay, in seconds, between frames. Smaller delays refresh the page faster. Fractional values are allowed.

If not specified, -delay will default to 1 second

-cookie, -target, -expires, -nph

These have the same meaning as the like-named parameters in `CGI::header()`.

If not specified, `-nph` will default to 1 (as needed for many servers, see below).

Heterogeneous Pages

Ordinarily all pages displayed by `CGI::Push` share a common MIME type. However by providing a value of "heterogeneous" or "dynamic" in the `do_push()` `-type` parameter, you can specify the MIME type of each page on a case-by-case basis.

If you use this option, you will be responsible for producing the HTTP header for each page. Simply modify your draw routine to look like this:

```
sub my_draw_routine {
    my($q,$counter) = @_;
    return header('text/html'),    # note we're producing the header here
           start_html('testing'),
           h1('testing'),
           "This page called $counter times";
}
```

You can add any header fields that you like, but some (cookies and status fields included) may not be interpreted by the browser. One interesting effect is to display a series of pages, then, after the last page, to redirect the browser to a new URL. Because `redirect()` does not work, the easiest way is with a `-refresh` header field, as shown below:

```
sub my_draw_routine {
    my($q,$counter) = @_;
    return undef if $counter > 10;
    return header('text/html'),    # note we're producing the header here
        start_html('testing'),
        h1('testing'),
        "This page called $counter times";
}

sub my_last_page {
    return header(-refresh=>'5; URL=http://somewhere.else/finished.html',
        -type=>'text/html'),
        start_html('Moved'),
        h1('This is the last page'),
        'Goodbye!',
        hr,
        end_html;
}
```

Changing the Page Delay on the Fly

If you would like to control the delay between pages on a page-by-page basis, call `push_delay()` from within your draw routine. `push_delay()` takes a single numeric argument representing the number of seconds you wish to delay after the current page is displayed and before displaying the next one. The delay may be fractional. Without parameters, `push_delay()` just returns the current delay.

INSTALLING CGI::Push SCRIPTS

Server push scripts must be installed as no-parsed-header (NPH) scripts in order to work correctly on many servers. On Unix systems, this is most often accomplished by prefixing the script's name with "nph-". Recognition of NPH scripts happens automatically with WebSTAR and Microsoft IIS. Users of other servers should see their documentation for help.

Apache web server from version 1.3b2 on does not need server push scripts installed as NPH scripts: the `-nph` parameter to `do_push()` may be set to a false value to disable the extra headers needed by an NPH script.

AUTHOR INFORMATION

Copyright 1995–1998, Lincoln D. Stein. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Address bug reports and comments to: lstein@cshl.org

BUGS

This section intentionally left blank.

SEE ALSO

CGI::Carp, *CGI*

NAME

CGI::Switch – Backward compatibility module for defunct CGI::Switch

SYNOPSIS

Do not use this module. It is deprecated.

ABSTRACT**DESCRIPTION****AUTHOR INFORMATION****BUGS****SEE ALSO**

NAME

CGI::Util – Internal utilities used by CGI module

SYNOPSIS

none

DESCRIPTION

no public subroutines

AUTHOR INFORMATION

Copyright 1995–1998, Lincoln D. Stein. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Address bug reports and comments to: lstein@cshl.org. When sending bug reports, please provide the version of CGI.pm, the version of Perl, the name and version of your Web server, and the name and version of the operating system you are using. If the problem is even remotely browser dependent, please provide information about the affected browsers as well.

SEE ALSO

[CGI](#)

NAME

charnames – define character names for `\N{named}` string literal escapes

SYNOPSIS

```
use charnames ':full';
print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";

use charnames ':short';
print "\N{greek:Sigma} is an upper-case sigma.\n";

use charnames qw(cyrillic greek);
print "\N{sigma} is Greek sigma, and \N{be} is Cyrillic b.\n";

print charnames::viacode(0x1234); # prints "ETHIOPIC SYLLABLE SEE"
printf "%04X", charnames::vianame("GOTHIC LETTER AHSA"); # prints "10330"
```

DESCRIPTION

Pragma `use charnames` supports arguments `:full`, `:short` and script names. If `:full` is present, for expansion of `\N{CHARNAME}` string `CHARNAME` is first looked in the list of standard Unicode names of chars. If `:short` is present, and `CHARNAME` has the form `SCRIPT:CNAME`, then `CNAME` is looked up as a letter in script `SCRIPT`. If pragma `use charnames` is used with script name arguments, then for `\N{CHARNAME}` the name `CHARNAME` is looked up as a letter in the given scripts (in the specified order).

For lookup of `CHARNAME` inside a given script `SCRIPTNAME` this pragma looks for the names

```
SCRIPTNAME CAPITAL LETTER CHARNAME
SCRIPTNAME SMALL LETTER CHARNAME
SCRIPTNAME LETTER CHARNAME
```

in the table of standard Unicode names. If `CHARNAME` is lowercase, then the `CAPITAL` variant is ignored, otherwise the `SMALL` variant is ignored.

Note that `\N{...}` is compile-time, it's a special form of string constant used inside double-quoted strings: in other words, you cannot use variables inside the `\N{...}`. If you want similar run-time functionality, use `charnames::vianame()`.

For the C0 and C1 control characters (U+0000..U+001F, U+0080..U+009F) as of Unicode 3.1, there are no official Unicode names but you can use instead the ISO 6429 names (LINE FEED, ESCAPE, and so forth). In Unicode 3.2 (as of Perl 5.8) some naming changes take place ISO 6429 has been updated, see [ALIASES](#). Also note that the U+UU80, U+0081, U+0084, and U+0099 do not have names even in ISO 6429.

Since the Unicode standard uses "U+HHHH", so can you: `\N{U+263a}` is the Unicode smiley face, or `\N{WHITE SMILING FACE}`.

CUSTOM TRANSLATORS

The mechanism of translation of `\N{...}` escapes is general and not hardwired into *charnames.pm*. A module can install custom translations (inside the scope which uses the module) with the following magic incantation:

```
use charnames ();          # for $charnames::hint_bits
sub import {
    shift;
    $^H |= $charnames::hint_bits;
    $^H{charnames} = \&translator;
}
```

Here `translator()` is a subroutine which takes `CHARNAME` as an argument, and returns text to insert into the string instead of the `\N{CHARNAME}` escape. Since the text to insert should be different in bytes mode and out of it, the function should check the current state of `bytes`-flag as in:

```

use bytes ();                                # for $bytes::hint_bits
sub translator {
    if ($^H & $bytes::hint_bits) {
        return bytes_translator(@_);
    }
    else {
        return utf8_translator(@_);
    }
}

```

charnames::viacode(code)

Returns the full name of the character indicated by the numeric code. The example

```
print charnames::viacode(0x2722);
```

prints "FOUR TEARDROP-SPOKED ASTERISK".

Returns undef if no name is known for the code.

This works only for the standard names, and does not yet apply to custom translators.

Notice that the name returned for U+FEFF is "ZERO WIDTH NO-BREAK SPACE", not "BYTE ORDER MARK".

charnames::vianame(name)

Returns the code point indicated by the name. The example

```
printf "%04X", charnames::vianame("FOUR TEARDROP-SPOKED ASTERISK");
```

prints "2722".

Returns undef if the name is unknown.

This works only for the standard names, and does not yet apply to custom translators.

ALIASES

A few aliases have been defined for convenience: instead of having to use the official names

```

LINE FEED (LF)
FORM FEED (FF)
CARRIAGE RETURN (CR)
NEXT LINE (NEL)

```

(yes, with parentheses) one can use

```

LINE FEED
FORM FEED
CARRIAGE RETURN
NEXT LINE
LF
FF
CR
NEL

```

One can also use

```

BYTE ORDER MARK
BOM

```

and

```

ZWNJ
ZWJ

```


for ZERO WIDTH NON-JOINER and ZERO WIDTH JOINER.

For backward compatibility one can use the old names for certain C0 and C1 controls

old	new
HORIZONTAL TABULATION	CHARACTER TABULATION
VERTICAL TABULATION	LINE TABULATION
FILE SEPARATOR	INFORMATION SEPARATOR FOUR
GROUP SEPARATOR	INFORMATION SEPARATOR THREE
RECORD SEPARATOR	INFORMATION SEPARATOR TWO
UNIT SEPARATOR	INFORMATION SEPARATOR ONE
PARTIAL LINE DOWN	PARTIAL LINE FORWARD
PARTIAL LINE UP	PARTIAL LINE BACKWARD

but the old names in addition to giving the character will also give a warning about being deprecated.

ILLEGAL CHARACTERS

If you ask by name for a character that does not exist, a warning is given and the Unicode *replacement character* "\x{FFFD}" is returned.

If you ask by code for a character that does not exist, no warning is given and `undef` is returned. (Though if you ask for a code point past U+10FFFF you do get a warning.)

BUGS

Since evaluation of the translation function happens in a middle of compilation (of a string literal), the translation function should not do any `evals` or `requires`. This restriction should be lifted in a future version of Perl.

NAME

Class::ISA — report the search path for a class's ISA tree

SYNOPSIS

```
# Suppose you go: use Food::Fishstick, and that uses and
# inherits from other things, which in turn use and inherit
# from other things. And suppose, for sake of brevity of
# example, that their ISA tree is the same as:

@Food::Fishstick::ISA = qw(Food::Fish Life::Fungus Chemicals);
@Food::Fish::ISA = qw(Food);
@Food::ISA = qw(Matter);
@Life::Fungus::ISA = qw(Life);
@Chemicals::ISA = qw(Matter);
@Life::ISA = qw(Matter);
@Matter::ISA = qw();

use Class::ISA;
print "Food::Fishstick path is:\n ",
      join(" ", Class::ISA::super_path('Food::Fishstick')),
      "\n";
```

That prints:

```
Food::Fishstick path is:
Food::Fish, Food, Matter, Life::Fungus, Life, Chemicals
```

DESCRIPTION

Suppose you have a class (like `Food::Fish::Fishstick`) that is derived, via its `@ISA`, from one or more superclasses (as `Food::Fish::Fishstick` is from `Food::Fish`, `Life::Fungus`, and `Chemicals`), and some of those superclasses may themselves each be derived, via its `@ISA`, from one or more superclasses (as above).

When, then, you call a method in that class (`$fishstick-calories`), Perl first searches there for that method, but if it's not there, it goes searching in its superclasses, and so on, in a depth-first (or maybe "height-first" is the word) search. In the above example, it'd first look in `Food::Fish`, then `Food`, then `Matter`, then `Life::Fungus`, then `Life`, then `Chemicals`.

This library, `Class::ISA`, provides functions that return that list — the list (in order) of names of classes Perl would search to find a method, with no duplicates.

FUNCTIONS

the function `Class::ISA::super_path($CLASS)`

This returns the ordered list of names of classes that Perl would search thru in order to find a method, with no duplicates in the list. `$CLASS` is not included in the list. `UNIVERSAL` is not included — if you need to consider it, add it to the end.

the function `Class::ISA::self_and_super_path($CLASS)`

Just like `super_path`, except that `$CLASS` is included as the first element.

the function `Class::ISA::self_and_super_versions($CLASS)`

This returns a hash whose keys are `$CLASS` and its (super-)superclasses, and whose values are the contents of each class's `$VERSION` (or undef, for classes with no `$VERSION`).

The code for `self_and_super_versions` is meant to serve as an example for precisely the kind of tasks I anticipate that `self_and_super_path` and `super_path` will be used for. You are strongly advised to read the source for `self_and_super_versions`, and the comments there.

CAUTIONARY NOTES

* Class::ISA doesn't export anything. You have to address the functions with a "Class::ISA::" on the front.

* Contrary to its name, Class::ISA isn't a class; it's just a package. Strange, isn't it?

* Say you have a loop in the ISA tree of the class you're calling one of the Class::ISA functions on: say that Food inherits from Matter, but Matter inherits from Food (for sake of argument). If Perl, while searching for a method, actually discovers this cyclicity, it will throw a fatal error. The functions in Class::ISA effectively ignore this cyclicity; the Class::ISA algorithm is "never go down the same path twice", and cyclicities are just a special case of that.

* The Class::ISA functions just look at @ISAs. But theoretically, I suppose, AUTOLOADs could bypass Perl's ISA-based search mechanism and do whatever they please. That would be bad behavior, tho; and I try not to think about that.

* If Perl can't find a method anywhere in the ISA tree, it then looks in the magical class UNIVERSAL. This is rarely relevant to the tasks that I expect Class::ISA functions to be put to, but if it matters to you, then instead of this:

```
@supers = Class::Tree::super_path($class);
```

do this:

```
@supers = (Class::Tree::super_path($class), 'UNIVERSAL');
```

And don't say no-one ever told ya!

* When you call them, the Class::ISA functions look at @ISAs anew — that is, there is no memoization, and so if ISAs change during runtime, you get the current ISA tree's path, not anything memoized. However, changing ISAs at runtime is probably a sign that you're out of your mind!

COPYRIGHT

Copyright (c) 1999, 2000 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

AUTHOR

Sean M. Burke sburke@cpan.org

NAME

Class::Struct – declare struct-like datatypes as Perl classes

SYNOPSIS

```
use Class::Struct;
    # declare struct, based on array:
struct( CLASS_NAME => [ ELEMENT_NAME => ELEMENT_TYPE, ... ] );
    # declare struct, based on hash:
struct( CLASS_NAME => { ELEMENT_NAME => ELEMENT_TYPE, ... } );

package CLASS_NAME;
use Class::Struct;
    # declare struct, based on array, implicit class name:
struct( ELEMENT_NAME => ELEMENT_TYPE, ... );

# Declare struct at compile time
use Class::Struct CLASS_NAME => [ ELEMENT_NAME => ELEMENT_TYPE, ... ];
use Class::Struct CLASS_NAME => { ELEMENT_NAME => ELEMENT_TYPE, ... };

package Myobj;
use Class::Struct;
    # declare struct with four types of elements:
struct( s => '$', a => '@', h => '%', c => 'My_Other_Class' );

$obj = new Myobj;                                # constructor

                                                # scalar type accessor:
$element_value = $obj->s;                        # element value
$obj->s('new value');                             # assign to element

                                                # array type accessor:
$array_ref = $obj->a;                             # reference to whole array
$array_element_value = $obj->a(2);                # array element value
$obj->a(2, 'new value');                           # assign to array element

                                                # hash type accessor:
$hash_ref = $obj->h;                             # reference to whole hash
$hash_element_value = $obj->h('x');               # hash element value
$obj->h('x', 'new value');                         # assign to hash element

                                                # class type accessor:
$element_value = $obj->c;                         # object reference
$obj->c->method(...);                             # call method of object
$obj->c(new My_Other_Class);                      # assign a new object
```

DESCRIPTION

Class::Struct exports a single function, `struct`. Given a list of element names and types, and optionally a class name, `struct` creates a Perl 5 class that implements a "struct-like" data structure.

The new class is given a constructor method, `new`, for creating struct objects.

Each element in the struct data has an accessor method, which is used to assign to the element and to fetch its value. The default accessor can be overridden by declaring a sub of the same name in the package. (See Example 2.)

Each element's type can be scalar, array, hash, or class.

The `struct()` function

The `struct` function has three forms of parameter-list.

```
struct( CLASS_NAME => [ ELEMENT_LIST ] );
struct( CLASS_NAME => { ELEMENT_LIST } );
struct( ELEMENT_LIST );
```

The first and second forms explicitly identify the name of the class being created. The third form assumes the current package name as the class name.

An object of a class created by the first and third forms is based on an array, whereas an object of a class created by the second form is based on a hash. The array-based forms will be somewhat faster and smaller; the hash-based forms are more flexible.

The class created by `struct` must not be a subclass of another class other than `UNIVERSAL`.

It can, however, be used as a superclass for other classes. To facilitate this, the generated constructor method uses a two-argument blessing. Furthermore, if the class is hash-based, the key of each element is prefixed with the class name (see *Perl Cookbook*, Recipe 13.12).

A function named `new` must not be explicitly defined in a class created by `struct`.

The *ELEMENT_LIST* has the form

```
NAME => TYPE, ...
```

Each name-type pair declares one element of the struct. Each element name will be defined as an accessor method unless a method by that name is explicitly defined; in the latter case, a warning is issued if the warning flag (`-w`) is set.

Class Creation at Compile Time

`Class::Struct` can create your class at compile time. The main reason for doing this is obvious, so your class acts like every other class in Perl. Creating your class at compile time will make the order of events similar to using any other class (or Perl module).

There is no significant speed gain between compile time and run time class creation, there is just a new, more standard order of events.

Element Types and Accessor Methods

The four element types — scalar, array, hash, and class — are represented by strings — ``$'`, ``@'`, ``%'`, and a class name — optionally preceded by a ``*'`.

The accessor method provided by `struct` for an element depends on the declared type of the element.

Scalar (``$'` or ``*$'`)

The element is a scalar, and by default is initialized to `undef` (but see [Initializing with new](#)).

The accessor's argument, if any, is assigned to the element.

If the element type is ``$'`, the value of the element (after assignment) is returned. If the element type is ``*$'`, a reference to the element is returned.

Array (``@'` or ``*@'`)

The element is an array, initialized by default to `()`.

With no argument, the accessor returns a reference to the element's whole array (whether or not the element was specified as ``@'` or ``*@'`).

With one or two arguments, the first argument is an index specifying one element of the array; the second argument, if present, is assigned to the array element. If the element type is ``@'`, the accessor returns the array element value. If the element type is ``*@'`, a reference to the array element is returned.

As a special case, when the accessor is called with an array reference as the sole argument, this causes an assignment of the whole array element. The object reference is returned.

Hash ('%' or '%')

The element is a hash, initialized by default to `()`.

With no argument, the accessor returns a reference to the element's whole hash (whether or not the element was specified as '%' or '%').

With one or two arguments, the first argument is a key specifying one element of the hash; the second argument, if present, is assigned to the hash element. If the element type is '%', the accessor returns the hash element value. If the element type is '%', a reference to the hash element is returned.

As a special case, when the accessor is called with a hash reference as the sole argument, this causes an assignment of the whole hash element. The object reference is returned.

Class ('Class_Name' or '*Class_Name')

The element's value must be a reference blessed to the named class or to one of its subclasses. The element is not initialized by default.

The accessor's argument, if any, is assigned to the element. The accessor will croak if this is not an appropriate object reference.

If the element type does not start with a '*', the accessor returns the element value (after assignment). If the element type starts with a '*', a reference to the element itself is returned.

Initializing with new

`struct` always creates a constructor called `new`. That constructor may take a list of initializers for the various elements of the new struct.

Each initializer is a pair of values: *element name* => *value*. The initializer value for a scalar element is just a scalar value. The initializer for an array element is an array reference. The initializer for a hash is a hash reference.

The initializer for a class element is an object of the corresponding class, or of one of its subclasses, or a reference to a hash containing named arguments to be passed to the element's constructor.

See Example 3 below for an example of initialization.

EXAMPLES**Example 1**

Giving a struct element a class type that is also a struct is how structs are nested. Here, `timeval` represents a time (seconds and microseconds), and `rusage` has two elements, each of which is of type `timeval`.

```
use Class::Struct;

struct( rusage => {
    ru_ftime => timeval, # seconds
    ru_stime => timeval, # microseconds
});

struct( timeval => [
    tv_secs => '$',
    tv_usecs => '$',
]);

# create an object:
my $t = new rusage;

# $t->ru_ftime and $t->ru_stime are objects of type timeval.
# set $t->ru_ftime to 100.0 sec and $t->ru_stime to 5.0 sec.
$t->ru_ftime->tv_secs(100);
$t->ru_ftime->tv_usecs(0);
```

```
$t->ru_stime->tv_secs(5);
$t->ru_stime->tv_usecs(0);
```

Example 2

An accessor function can be redefined in order to provide additional checking of values, etc. Here, we want the `count` element always to be nonnegative, so we redefine the `count` accessor accordingly.

```
package MyObj;
use Class::Struct;

# declare the struct
struct ( 'MyObj', { count => '$', stuff => '%' } );

# override the default accessor method for 'count'
sub count {
    my $self = shift;
    if ( @_ ) {
        die 'count must be nonnegative' if $_[0] < 0;
        $self->{'count'} = shift;
        warn "Too many args to count" if @_;
    }
    return $self->{'count'};
}

package main;
$x = new MyObj;
print "\$x->count(5) = ", $x->count(5), "\n";
# prints '$x->count(5) = 5'

print "\$x->count = ", $x->count, "\n";
# prints '$x->count = 5'

print "\$x->count(-5) = ", $x->count(-5), "\n";
# dies due to negative argument!
```

Example 3

The constructor of a generated class can be passed a list of *element=value* pairs, with which to initialize the struct. If no initializer is specified for a particular element, its default initialization is performed instead. Initializers for non-existent elements are silently ignored.

Note that the initializer for a nested class may be specified as an object of that class, or as a reference to a hash of initializers that are passed on to the nested struct's constructor.

```
use Class::Struct;

struct Breed =>
{
    name    => '$',
    cross   => '$',
};

struct Cat =>
[
    name      => '$',
    kittens   => '@',
    markings  => '%',
    breed     => 'Breed',
];

my $cat = Cat->new( name    => 'Socks',
                   kittens => ['Monica', 'Kenneth'],
```

```

        markings => { socks=>1, blaze=>"white" },
        breed    => Breed->new(name=>'short-hair', cross=>1),
    or: breed    => {name=>'short-hair', cross=>1},
    );

    print "Once a cat called ", $cat->name, "\n";
    print "(which was a ", $cat->breed->name, ")\n";
    print "had two kittens: ", join(' and ', @{$cat->kittens}), "\n";

```

Author and Modification History

Modified by Damian Conway, 2001-09-10, v0.62.

Modified implicit construction of nested objects.
 Now will also take an object ref instead of requiring a hash ref.
 Also default initializes nested object attributes to undef, rather than calling object constructor without args

Original over-helpfulness was fraught with problems:

- * the class's constructor might not be called 'new'
- * the class might not have a hash-like-arguments constructor
- * the class might not have a no-argument constructor
- * "recursive" data structures didn't work well:

```

package Person;
struct { mother => 'Person', father => 'Person' };

```

Modified by Casey West, 2000-11-08, v0.59.

Added the ability for compile time class creation.

Modified by Damian Conway, 1999-03-05, v0.58.

Added handling of hash-like arg list to class ctor.

Changed to two-argument blessing in ctor to support derivation from created classes.

Added classname prefixes to keys in hash-based classes (refer to "Perl Cookbook", Recipe 13.12 for rationale).

Corrected behaviour of accessors for '*@' and '*%' struct elements. Package now implements documented behaviour when returning a reference to an entire hash or array element. Previously these were returned as a reference to a reference to the element.

Renamed to `Class::Struct` and modified by Jim Miner, 1997-04-02.

```

members() function removed.
Documentation corrected and extended.
Use of struct() in a subclass prohibited.
User definition of accessor allowed.
Treatment of '*' in element types corrected.
Treatment of classes as element types corrected.
Class name to struct() made optional.
Diagnostic checks added.

```

Originally `Class::Template` by Dean Roehrich.

```

# Template.pm --- struct/member template builder
# 12mar95
# Dean Roehrich
#

```



```
# changes/bugs fixed since 28nov94 version:
#   - podified
# changes/bugs fixed since 21nov94 version:
#   - Fixed examples.
# changes/bugs fixed since 02sep94 version:
#   - Moved to Class::Template.
# changes/bugs fixed since 20feb94 version:
#   - Updated to be a more proper module.
#   - Added "use strict".
#   - Bug in build_methods, was using @var when @$var needed.
#   - Now using my() rather than local().
#
# Uses perl5 classes to create nested data types.
# This is offered as one implementation of Tom Christiansen's "structs.pl"
# idea.
```

NAME

constant – Perl pragma to declare constants

SYNOPSIS

```
use constant PI      => 4 * atan2(1, 1);
use constant DEBUG => 0;

print "Pi equals ", PI, "...\\n" if DEBUG;

use constant {
    SEC  => 0,
    MIN  => 1,
    HOUR => 2,
    MDAY => 3,
    MON  => 4,
    YEAR => 5,
    WDAY => 6,
    YDAY => 7,
    ISDST => 8,
};

use constant WEEKDAYS => qw(
    Sunday Monday Tuesday Wednesday Thursday Friday Saturday
);

print "Today is ", (WEEKDAYS)[ (localtime)[WDAY] ], "...\\n";
```

DESCRIPTION

This will declare a symbol to be a constant with the given value.

When you declare a constant such as PI using the method shown above, each machine your script runs upon can have as many digits of accuracy as it can use. Also, your program will be easier to read, more likely to be maintained (and maintained correctly), and far less likely to send a space probe to the wrong planet because nobody noticed the one equation in which you wrote 3.14195.

When a constant is used in an expression, perl replaces it with its value at compile time, and may then optimize the expression further. In particular, any code in an if (CONSTANT) block will be optimized away if the constant is false.

NOTES

As with all use directives, defining a constant happens at compile time. Thus, it's probably not correct to put a constant declaration inside of a conditional statement (like if (\$foo) { use constant ... }).

Constants defined using this module cannot be interpolated into strings like variables. However, concatenation works just fine:

```
print "Pi equals PI...\\n";      # WRONG: does not expand "PI"
print "Pi equals ".PI."...\\n";  # right
```

Even though a reference may be declared as a constant, the reference may point to data which may be changed, as this code shows.

```
use constant ARRAY => [ 1,2,3,4 ];
print ARRAY->[1];
ARRAY->[1] = " be changed";
print ARRAY->[1];
```

Dereferencing constant references incorrectly (such as using an array subscript on a constant hash reference, or vice versa) will be trapped at compile time.

Constants belong to the package they are defined in. To refer to a constant defined in another package, specify the full package name, as in `Some::Package::CONSTANT`. Constants may be exported by modules, and may also be called as either class or instance methods, that is, as `<Some::Package-CONSTANT` or as `< $obj-CONSTANT` where `$obj` is an instance of `Some::Package`. Subclasses may define their own constants to override those in their base class.

The use of all caps for constant names is merely a convention, although it is recommended in order to make constants stand out and to help avoid collisions with other barewords, keywords, and subroutine names. Constant names must begin with a letter or underscore. Names beginning with a double underscore are reserved. Some poor choices for names will generate warnings, if warnings are enabled at compile time.

List constants

Constants may be lists of more (or less) than one value. A constant with no values evaluates to `undef` in scalar context. Note that constants with more than one value do *not* return their last value in scalar context as one might expect. They currently return the number of values, but **this may change in the future**. Do not use constants with multiple values in scalar context.

NOTE: This implies that the expression defining the value of a constant is evaluated in list context. This may produce surprises:

```
use constant TIMESTAMP => localtime;           # WRONG!
use constant TIMESTAMP => scalar localtime;     # right
```

The first line above defines `TIMESTAMP` as a 9-element list, as returned by `localtime()` in list context. To set it to the string returned by `localtime()` in scalar context, an explicit `scalar` keyword is required.

List constants are lists, not arrays. To index or slice them, they must be placed in parentheses.

```
my @workdays = WEEKDAYS[1 .. 5];               # WRONG!
my @workdays = (WEEKDAYS)[1 .. 5];             # right
```

Defining multiple constants at once

Instead of writing multiple `use constant` statements, you may define multiple constants in a single statement by giving, instead of the constant name, a reference to a hash where the keys are the names of the constants to be defined. Obviously, all constants defined using this method must have a single value.

```
use constant {
    FOO => "A single value",
    BAR => "This", "won't", "work!",           # Error!
};
```

This is a fundamental limitation of the way hashes are constructed in Perl. The error messages produced when this happens will often be quite cryptic — in the worst case there may be none at all, and you'll only later find that something is broken.

When defining multiple constants, you cannot use the values of other constants defined in the same declaration. This is because the calling package doesn't know about any constant within that group until *after* the `use` statement is finished.

```
use constant {
    BITMASK => 0xAFBAEBA8,
    NEGMASK => ~BITMASK,                       # Error!
};
```

Magic constants

Magical values and references can be made into constants at compile time, allowing for way cool stuff like this. (These error numbers aren't totally portable, alas.)

```
use constant E2BIG => ($! = 7);
print E2BIG, "\n";           # something like "Arg list too long"
```

```
print 0+E2BIG, "\n";           # "7"
```

You can't produce a tied constant by giving a tied scalar as the value. References to tied variables, however, can be used as constants without any problems.

TECHNICAL NOTES

In the current implementation, scalar constants are actually inlinable subroutines. As of version 5.004 of Perl, the appropriate scalar constant is inserted directly in place of some subroutine calls, thereby saving the overhead of a subroutine call. See *Constant Functions in perlsyn* for details about how and when this happens.

In the rare case in which you need to discover at run time whether a particular constant has been declared via this module, you may use this function to examine the hash `%constant::declared`. If the given constant name does not include a package name, the current package is used.

```
sub declared ($) {
    use constant 1.01;           # don't omit this!
    my $name = shift;
    $name =~ s/^\s*/main::/;
    my $pkg = caller;
    my $full_name = $name =~ /\s*? $name : "${pkg}::$name";
    $constant::declared{$full_name};
}
```

BUGS

In the current version of Perl, list constants are not inlined and some symbols may be redefined without generating a warning.

It is not possible to have a subroutine or a keyword with the same name as a constant in the same package. This is probably a Good Thing.

A constant with a name in the list `STDIN STDOUT STDERR ARGV ARGVOUT ENV INC SIG` is not allowed anywhere but in package `main::`, for technical reasons.

Unlike constants in some languages, these cannot be overridden on the command line or via environment variables.

You can get into trouble if you use constants in a context which automatically quotes barewords (as is true for any subroutine call). For example, you can't say `$hash{CONSTANT}` because `CONSTANT` will be interpreted as a string. Use `$hash{CONSTANT() }` or `$hash{+CONSTANT}` to prevent the bareword quoting mechanism from kicking in. Similarly, since the `< =` operator quotes a bareword immediately to its left, you have to say `< CONSTANT() = 'value'` (or simply use a comma in place of the big arrow) instead of `< CONSTANT = 'value'`.

AUTHOR

Tom Phoenix, <rootbeer@redcat.com>, with help from many other folks.

Multiple constant declarations at once added by Casey West, <casey@geeknest.com>.

Documentation mostly rewritten by Ilmari Karonen, <perl@itz.pp.sci.fi>.

COPYRIGHT

Copyright (C) 1997, 1999 Tom Phoenix

This module is free software; you can redistribute it or modify it under the same terms as Perl itself.

NAME

CPAN – query, download and build perl modules from CPAN sites

SYNOPSIS

Interactive mode:

```
perl -MCPAN -e shell;
```

Batch mode:

```
use CPAN;

autobundle, clean, install, make, recompile, test
```

DESCRIPTION

The CPAN module is designed to automate the make and install of perl modules and extensions. It includes some searching capabilities and knows how to use Net::FTP or LWP (or lynx or an external ftp client) to fetch the raw data from the net.

Modules are fetched from one or more of the mirrored CPAN (Comprehensive Perl Archive Network) sites and unpacked in a dedicated directory.

The CPAN module also supports the concept of named and versioned *bundles* of modules. Bundles simplify the handling of sets of related modules. See Bundles below.

The package contains a session manager and a cache manager. There is no status retained between sessions. The session manager keeps track of what has been fetched, built and installed in the current session. The cache manager keeps track of the disk space occupied by the make processes and deletes excess space according to a simple FIFO mechanism.

For extended searching capabilities there's a plugin for CPAN available, [CPAN:::WAIT/CPAN:::WAIT](#). CPAN:::WAIT is a full-text search engine that indexes all documents available in CPAN authors directories. If CPAN:::WAIT is installed on your system, the interactive shell of CPAN.pm will enable the `wq`, `wr`, `wd`, `wl`, and `wh` commands which send queries to the WAIT server that has been configured for your installation.

All other methods provided are accessible in a programmer style and in an interactive shell style.

Interactive Mode

The interactive mode is entered by running

```
perl -MCPAN -e shell
```

which puts you into a readline interface. You will have the most fun if you install Term::ReadKey and Term::ReadLine to enjoy both history and command completion.

Once you are on the command line, type 'h' and the rest should be self-explanatory.

The function call `shell` takes two optional arguments, one is the prompt, the second is the default initial command line (the latter only works if a real ReadLine interface module is installed).

The most common uses of the interactive modes are

Searching for authors, bundles, distribution files and modules

There are corresponding one-letter commands `a`, `b`, `d`, and `m` for each of the four categories and another, `i` for any of the mentioned four. Each of the four entities is implemented as a class with slightly differing methods for displaying an object.

Arguments you pass to these commands are either strings exactly matching the identification string of an object or regular expressions that are then matched case-insensitively against various attributes of the objects. The parser recognizes a regular expression only if you enclose it between two slashes.

The principle is that the number of found objects influences how an item is displayed. If the search finds

one item, the result is displayed with the rather verbose method `as_string`, but if we find more than one, we display each object with the terse method `<as_glimpse>`.

make, test, install, clean modules or distributions

These commands take any number of arguments and investigate what is necessary to perform the action. If the argument is a distribution file name (recognized by embedded slashes), it is processed. If it is a module, CPAN determines the distribution file in which this module is included and processes that, following any dependencies named in the module's `Makefile.PL` (this behavior is controlled by *prerequisites_policy*.)

Any `make` or `test` are run unconditionally. An

```
install <distribution_file>
```

also is run unconditionally. But for

```
install <module>
```

CPAN checks if an install is actually needed for it and prints *module up to date* in the case that the distribution file containing the module doesn't need to be updated.

CPAN also keeps track of what it has done within the current session and doesn't try to build a package a second time regardless if it succeeded or not. The `force` command takes as a first argument the method to invoke (currently: `make`, `test`, or `install`) and executes the command from scratch.

Example:

```
cpan> install OpenGL
OpenGL is up to date.
cpan> force install OpenGL
Running make
OpenGL-0.4/
OpenGL-0.4/COPYRIGHT
[...]
```

A `clean` command results in a

```
make clean
```

being executed within the distribution file's working directory.

get, readme, look module or distribution

`get` downloads a distribution file without further action. `readme` displays the `README` file of the associated distribution. `look` gets and untars (if not yet done) the distribution file, changes to the appropriate directory and opens a subshell process in that directory.

ls author

`ls` lists all distribution files in and below an author's CPAN directory. Only those files that contain modules are listed and if there is more than one for any given module, only the most recent one is listed.

Signals

`CPAN.pm` installs signal handlers for `SIGINT` and `SIGTERM`. While you are in the `cpan-shell` it is intended that you can press `^C` anytime and return to the `cpan-shell` prompt. A `SIGTERM` will cause the `cpan-shell` to clean up and leave the shell loop. You can emulate the effect of a `SIGTERM` by sending two consecutive `SIGINT`s, which usually means by pressing `^C` twice.

`CPAN.pm` ignores a `SIGPIPE`. If the user sets `inactivity_timeout`, a `SIGALRM` is used during the run of the `perl Makefile.PL` subprocess.

CPAN::Shell

The commands that are available in the shell interface are methods in the package `CPAN::Shell`. If you enter the shell command, all your input is split by the `Text::ParseWords::shellwords()` routine which

acts like most shells do. The first word is being interpreted as the method to be called and the rest of the words are treated as arguments to this method. Continuation lines are supported if a line ends with a literal backslash.

autobundle

`autobundle` writes a bundle file into the `$CPAN::Config->{cpan_home}/Bundle` directory. The file contains a list of all modules that are both available from CPAN and currently installed within `@INC`. The name of the bundle file is based on the current date and a counter.

recompile

`recompile()` is a very special command in that it takes no argument and runs the `make/test/install` cycle with brute force over all installed dynamically loadable extensions (aka XS modules) with `'force'` in effect. The primary purpose of this command is to finish a network installation. Imagine, you have a common source tree for two different architectures. You decide to do a completely independent fresh installation. You start on one architecture with the help of a Bundle file produced earlier. CPAN installs the whole Bundle for you, but when you try to repeat the job on the second architecture, CPAN responds with a "Foo up to date" message for all modules. So you invoke CPAN's `recompile` on the second architecture and you're done.

Another popular use for `recompile` is to act as a rescue in case your perl breaks binary compatibility. If one of the modules that CPAN uses is in turn depending on binary compatibility (so you cannot run CPAN commands), then you should try the `CPAN::Nox` module for recovery.

The four CPAN::* Classes: Author, Bundle, Module, Distribution

Although it may be considered internal, the class hierarchy does matter for both users and programmer. CPAN.pm deals with above mentioned four classes, and all those classes share a set of methods. A classical single polymorphism is in effect. A metaclass object registers all objects of all kinds and indexes them with a string. The strings referencing objects have a separated namespace (well, not completely separated):

Namespace	Class
words containing a "/" (slash)	Distribution
words starting with <code>Bundle::</code>	Bundle
everything else	Module or Author

Modules know their associated Distribution objects. They always refer to the most recent official release. Developers may mark their releases as unstable development versions (by inserting an underbar into the module version number which will also be reflected in the distribution name when you run `'make dist'`), so the really hottest and newest distribution is not always the default. If a module `Foo` circulates on CPAN in both version 1.23 and 1.23_90, CPAN.pm offers a convenient way to install version 1.23 by saying

```
install Foo
```

This would install the complete distribution file (say `BAR/Foo-1.23.tar.gz`) with all accompanying material. But if you would like to install version 1.23_90, you need to know where the distribution file resides on CPAN relative to the `authors/id/` directory. If the author is `BAR`, this might be `BAR/Foo-1.23_90.tar.gz`; so you would have to say

```
install BAR/Foo-1.23_90.tar.gz
```

The first example will be driven by an object of the class `CPAN::Module`, the second by an object of class `CPAN::Distribution`.

Programmer's interface

If you do not enter the shell, the available shell commands are both available as methods (`CPAN::Shell->install(...)`) and as functions in the calling package (`install(...)`).

There's currently only one class that has a stable interface – `CPAN::Shell`. All commands that are available in the CPAN shell are methods of the class `CPAN::Shell`. Each of the commands that produce listings of modules (`r`, `autobundle`, `u`) also return a list of the IDs of all modules within the list.

expand(\$type,@things)

The IDs of all objects available within a program are strings that can be expanded to the corresponding real objects with the `CPAN::Shell->expand("Module",@things)` method. Expand returns a list of `CPAN::Module` objects according to the `@things` arguments given. In scalar context it only returns the first element of the list.

expandany(@things)

Like `expand`, but returns objects of the appropriate type, i.e. `CPAN::Bundle` objects for bundles, `CPAN::Module` objects for modules and `CPAN::Distribution` objects for distributions.

Programming Examples

This enables the programmer to do operations that combine functionalities that are available in the shell.

```
# install everything that is outdated on my disk:
perl -MCPAN -e 'CPAN::Shell->install(CPAN::Shell->r)'

# install my favorite programs if necessary:
for $mod (qw(Net::FTP Digest::MD5 Data::Dumper)) {
    my $obj = CPAN::Shell->expand('Module', $mod);
    $obj->install;
}

# list all modules on my disk that have no VERSION number
for $mod (CPAN::Shell->expand("Module", "/./")) {
    next unless $mod->inst_file;
    # MakeMaker convention for undefined $VERSION:
    next unless $mod->inst_version eq "undef";
    print "No VERSION in ", $mod->id, "\n";
}

# find out which distribution on CPAN contains a module:
print CPAN::Shell->expand("Module", "Apache::Constants")->cpan_file
```

Or if you want to write a cronjob to watch The CPAN, you could list all modules that need updating. First a quick and dirty way:

```
perl -e 'use CPAN; CPAN::Shell->r;'
```

If you don't want to get any output in the case that all modules are up to date, you can parse the output of above command for the regular expression `//modules are up to date//` and decide to mail the output only if it doesn't match. Ick?

If you prefer to do it more in a programmer style in one single process, maybe something like this suits you better:

```
# list all modules on my disk that have newer versions on CPAN
for $mod (CPAN::Shell->expand("Module", "/./")) {
    next unless $mod->inst_file;
    next if $mod->uptodate;
    printf "Module %s is installed as %s, could be updated to %s from CPAN\n",
        $mod->id, $mod->inst_version, $mod->cpan_version;
}
```

If that gives you too much output every day, you maybe only want to watch for three modules. You can write

```
for $mod (CPAN::Shell->expand("Module", "/Apache|LWP|CGI/")) {
```

as the first line instead. Or you can combine some of the above tricks:


```
# watch only for a new mod_perl module
$mod = CPAN::Shell->expand("Module","mod_perl");
exit if $mod->uptodate;
# new mod_perl arrived, let me know all update recommendations
CPAN::Shell->r;
```

Methods in the other Classes

The programming interface for the classes `CPAN::Module`, `CPAN::Distribution`, `CPAN::Bundle`, and `CPAN::Author` is still considered beta and partially even alpha. In the following paragraphs only those methods are documented that have proven useful over a longer time and thus are unlikely to change.

`CPAN::Author::as_glimpse()`

Returns a one-line description of the author

`CPAN::Author::as_string()`

Returns a multi-line description of the author

`CPAN::Author::email()`

Returns the author's email address

`CPAN::Author::fullname()`

Returns the author's name

`CPAN::Author::name()`

An alias for `fullname`

`CPAN::Bundle::as_glimpse()`

Returns a one-line description of the bundle

`CPAN::Bundle::as_string()`

Returns a multi-line description of the bundle

`CPAN::Bundle::clean()`

Recursively runs the `clean` method on all items contained in the bundle.

`CPAN::Bundle::contains()`

Returns a list of objects' IDs contained in a bundle. The associated objects may be bundles, modules or distributions.

`CPAN::Bundle::force($method,@args)`

Forces CPAN to perform a task that normally would have failed. Force takes as arguments a method name to be called and any number of additional arguments that should be passed to the called method. The internals of the object get the needed changes so that `CPAN.pm` does not refuse to take the action. The force is passed recursively to all contained objects.

`CPAN::Bundle::get()`

Recursively runs the `get` method on all items contained in the bundle

`CPAN::Bundle::inst_file()`

Returns the highest installed version of the bundle in either `@INC` or `$CPAN::Config->{cpan_home}`. Note that this is different from `CPAN::Module::inst_file`.

`CPAN::Bundle::inst_version()`

Like `CPAN::Bundle::inst_file`, but returns the `$VERSION`

`CPAN::Bundle::uptodate()`

Returns 1 if the bundle itself and all its members are uptodate.

CPAN::Bundle::install()

Recursively runs the `install` method on all items contained in the bundle

CPAN::Bundle::make()

Recursively runs the `make` method on all items contained in the bundle

CPAN::Bundle::readme()

Recursively runs the `readme` method on all items contained in the bundle

CPAN::Bundle::test()

Recursively runs the `test` method on all items contained in the bundle

CPAN::Distribution::as_glimpse()

Returns a one-line description of the distribution

CPAN::Distribution::as_string()

Returns a multi-line description of the distribution

CPAN::Distribution::clean()

Changes to the directory where the distribution has been unpacked and runs `make clean` there.

CPAN::Distribution::containsmods()

Returns a list of IDs of modules contained in a distribution file. Only works for distributions listed in the `02packages.details.txt.gz` file. This typically means that only the most recent version of a distribution is covered.

CPAN::Distribution::cvs_import()

Changes to the directory where the distribution has been unpacked and runs something like

```
cvs -d $cvs_root import -m $cvs_log $cvs_dir $userid v$version
there.
```

CPAN::Distribution::dir()

Returns the directory into which this distribution has been unpacked.

CPAN::Distribution::force(\$method,@args)

Forces CPAN to perform a task that normally would have failed. Force takes as arguments a method name to be called and any number of additional arguments that should be passed to the called method. The internals of the object get the needed changes so that CPAN.pm does not refuse to take the action.

CPAN::Distribution::get()

Downloads the distribution from CPAN and unpacks it. Does nothing if the distribution has already been downloaded and unpacked within the current session.

CPAN::Distribution::install()

Changes to the directory where the distribution has been unpacked and runs the external command `make install` there. If `make` has not yet been run, it will be run first. A `make test` will be issued in any case and if this fails, the install will be canceled. The cancellation can be avoided by letting `force` run the `install` for you.

CPAN::Distribution::isa_perl()

Returns 1 if this distribution file seems to be a perl distribution. Normally this is derived from the file name only, but the index from CPAN can contain a hint to achieve a return value of true for other filenames too.

CPAN::Distribution::look()

Changes to the directory where the distribution has been unpacked and opens a subshell there. Exiting the subshell returns.

CPAN::Distribution::make()

First runs the `get` method to make sure the distribution is downloaded and unpacked. Changes to the directory where the distribution has been unpacked and runs the external commands `perl Makefile.PL` and `make` there.

CPAN::Distribution::prereq_pm()

Returns the hash reference that has been announced by a distribution as the `PREREQ_PM` hash in the `Makefile.PL`. Note: works only after an attempt has been made to make the distribution. Returns `undef` otherwise.

CPAN::Distribution::readme()

Downloads the `README` file associated with a distribution and runs it through the pager specified in `$CPAN::Config->{pager}`.

CPAN::Distribution::test()

Changes to the directory where the distribution has been unpacked and runs `make test` there.

CPAN::Distribution::uptodate()

Returns 1 if all the modules contained in the distribution are uptodate. Relies on `containsmods`.

CPAN::Index::force_reload()

Forces a reload of all indices.

CPAN::Index::reload()

Reloads all indices if they have been read more than `$CPAN::Config->{index_expire}` days.

CPAN::InfoObj::dump()

`CPAN::Author`, `CPAN::Bundle`, `CPAN::Module`, and `CPAN::Distribution` inherit this method. It prints the data structure associated with an object. Useful for debugging. Note: the data structure is considered internal and thus subject to change without notice.

CPAN::Module::as_glimpse()

Returns a one-line description of the module

CPAN::Module::as_string()

Returns a multi-line description of the module

CPAN::Module::clean()

Runs a clean on the distribution associated with this module.

CPAN::Module::cpan_file()

Returns the filename on CPAN that is associated with the module.

CPAN::Module::cpan_version()

Returns the latest version of this module available on CPAN.

CPAN::Module::cvs_import()

Runs a `cvs_import` on the distribution associated with this module.

CPAN::Module::description()

Returns a 44 character description of this module. Only available for modules listed in The Module List (`CPAN/modules/00modlist.long.html` or `00modlist.long.txt.gz`)

CPAN::Module::force(\$method,@args)

Forces CPAN to perform a task that normally would have failed. `Force` takes as arguments a method name to be called and any number of additional arguments that should be passed to the called method. The internals of the object get the needed changes so that `CPAN.pm` does not refuse to take the action.

CPAN::Module::get()

Runs a `get` on the distribution associated with this module.

CPAN::Module::inst_file()

Returns the filename of the module found in `@INC`. The first file found is reported just like `perl` itself stops searching `@INC` when it finds a module.

CPAN::Module::inst_version()

Returns the version number of the module in readable format.

CPAN::Module::install()

Runs an `install` on the distribution associated with this module.

CPAN::Module::look()

Changes to the directory where the distribution associated with this module has been unpacked and opens a subshell there. Exiting the subshell returns.

CPAN::Module::make()

Runs a `make` on the distribution associated with this module.

CPAN::Module::manpage_headline()

If module is installed, peeks into the module's manpage, reads the headline and returns it. Moreover, if the module has been downloaded within this session, does the equivalent on the downloaded module even if it is not installed.

CPAN::Module::readme()

Runs a `readme` on the distribution associated with this module.

CPAN::Module::test()

Runs a `test` on the distribution associated with this module.

CPAN::Module::uptodate()

Returns 1 if the module is installed and up-to-date.

CPAN::Module::userid()

Returns the author's ID of the module.

Cache Manager

Currently the cache manager only keeps track of the build directory (`$CPAN::Config->{build_dir}`). It is a simple FIFO mechanism that deletes complete directories below `build_dir` as soon as the size of all directories there gets bigger than `$CPAN::Config->{build_cache}` (in MB). The contents of this cache may be used for later re-installations that you intend to do manually, but will never be trusted by CPAN itself. This is due to the fact that the user might use these directories for building modules on different architectures.

There is another directory (`$CPAN::Config->{keep_source_where}`) where the original distribution files are kept. This directory is not covered by the cache manager and must be controlled by the user. If you choose to have the same directory as `build_dir` and as `keep_source_where` directory, then your sources will be deleted with the same fifo mechanism.

Bundles

A bundle is just a perl module in the namespace `Bundle::` that does not define any functions or methods. It usually only contains documentation.

It starts like a perl module with a package declaration and a `$VERSION` variable. After that the pod section looks like any other pod with the only difference being that *one special pod section* exists starting with (verbatim):

```
=head1 CONTENTS
```

In this pod section each line obeys the format

```
Module_Name [Version_String] [- optional text]
```

The only required part is the first field, the name of a module (e.g. `Foo::Bar`, ie. *not* the name of the distribution file). The rest of the line is optional. The comment part is delimited by a dash just as in the man page header.

The distribution of a bundle should follow the same convention as other distributions.

Bundles are treated specially in the CPAN package. If you say 'install Bundle::Tkkit' (assuming such a bundle exists), CPAN will install all the modules in the CONTENTS section of the pod. You can install your own Bundles locally by placing a conformant Bundle file somewhere into your @INC path. The `autobundle()` command which is available in the shell interface does that for you by including all currently installed modules in a snapshot bundle file.

Prerequisites

If you have a local mirror of CPAN and can access all files with "file:" URLs, then you only need a perl better than perl5.003 to run this module. Otherwise Net::FTP is strongly recommended. LWP may be required for non-UNIX systems or if your nearest CPAN site is associated with a URL that is not `ftp:`.

If you have neither Net::FTP nor LWP, there is a fallback mechanism implemented for an external `ftp` command or for an external `lynx` command.

Finding packages and VERSION

This module presumes that all packages on CPAN

- declare their \$VERSION variable in an easy to parse manner. This prerequisite can hardly be relaxed because it consumes far too much memory to load all packages into the running program just to determine the \$VERSION variable. Currently all programs that are dealing with version use something like this

```
perl -MExtUtils::MakeMaker -le \
    'print MM->parse_version(shift)' filename
```

If you are author of a package and wonder if your \$VERSION can be parsed, please try the above method.

- come as compressed or gzipped tarfiles or as zip files and contain a Makefile.PL (well, we try to handle a bit more, but without much enthusiasm).

Debugging

The debugging of this module is a bit complex, because we have interferences of the software producing the indices on CPAN, of the mirroring process on CPAN, of packaging, of configuration, of synchronicity, and of bugs within CPAN.pm.

For code debugging in interactive mode you can try "o debug" which will list options for debugging the various parts of the code. You should know that "o debug" has built-in completion support.

For data debugging there is the `dump` command which takes the same arguments as `make/test/install` and outputs the object's `Data::Dumper dump`.

Floppy, Zip, Offline Mode

CPAN.pm works nicely without network too. If you maintain machines that are not networked at all, you should consider working with file: URLs. Of course, you have to collect your modules somewhere first. So you might use CPAN.pm to put together all you need on a networked machine. Then copy the `$CPAN::Config-{keep_source_where}` (but not `$CPAN::Config-{build_dir}`) directory on a floppy. This floppy is kind of a personal CPAN. CPAN.pm on the non-networked machines works nicely with this floppy. See also below the paragraph about CD-ROM support.

CONFIGURATION

When the CPAN module is installed, a site wide configuration file is created as CPAN/Config.pm. The default values defined there can be overridden in another configuration file: CPAN/MyConfig.pm. You can store this file in \$HOME/.cpan/CPAN/MyConfig.pm if you want, because \$HOME/.cpan is added to the search path of the CPAN module before the use() or require() statements.

Currently the following keys in the hash reference \$CPAN::Config are defined:

build_cache	size of cache for directories to build modules
build_dir	locally accessible directory to build modules
index_expire	after this many days refetch index files
cache_metadata	use serializer to cache metadata
cpan_home	local directory reserved for this package
dontload_hash	anonymous hash: modules in the keys will not be loaded by the CPAN::has_inst() routine
gzip	location of external program gzip
inactivity_timeout	breaks interactive Makefile.PLs after this many seconds inactivity. Set to 0 to never break.
inhibit_startup_message	if true, does not print the startup message
keep_source_where	directory in which to keep the source (if we do)
make	location of external make program
make_arg	arguments that should always be passed to 'make'
make_install_arg	same as make_arg for 'make install'
makepl_arg	arguments passed to 'perl Makefile.PL'
pager	location of external program more (or any pager)
prerequisites_policy	what to do if you are missing module prerequisites ('follow' automatically, 'ask' me, or 'ignore')
proxy_user	username for accessing an authenticating proxy
proxy_pass	password for accessing an authenticating proxy
scan_cache	controls scanning of cache ('atstart' or 'never')
tar	location of external program tar
term_is_latin	if true internal UTF-8 is translated to ISO-8859-1 (and nonsense for characters outside latin range)
unzip	location of external program unzip
urllist	arrayref to nearby CPAN sites (or equivalent locations)
wait_list	arrayref to a wait server to try (See CPAN::WAIT)
ftp_proxy,	} the three usual variables for configuring
http_proxy,	} proxy requests. Both as CPAN::Config variables
no_proxy	} and as environment variables configurable.

You can set and query each of these options interactively in the cpan shell with the command set defined within the o conf command:

- o conf <scalar option>
prints the current value of the *scalar option*
- o conf <scalar option> <value>
Sets the value of the *scalar option* to *value*
- o conf <list option>
prints the current value of the *list option* in MakeMaker's neatvalue format.
- o conf <list option> [shift|pop]
shifts or pops the array in the *list option* variable

- o `conf <list option> [unshift|push|splice] <list>`

works like the corresponding perl commands.

Note on urllist parameter's format

urllist parameters are URLs according to RFC 1738. We do a little guessing if your URL is not compliant, but if you have problems with file URLs, please try the correct format. Either:

```
file://localhost/whatever/ftp/pub/CPAN/
```

or

```
file:///home/ftp/pub/CPAN/
```

urllist parameter has CD-ROM support

The urllist parameter of the configuration table contains a list of URLs that are to be used for downloading. If the list contains any file URLs, CPAN always tries to get files from there first. This feature is disabled for index files. So the recommendation for the owner of a CD-ROM with CPAN contents is: include your local, possibly outdated CD-ROM as a file URL at the end of urllist, e.g.

- o `conf urllist push file://localhost/CDROM/CPAN`

CPAN.pm will then fetch the index files from one of the CPAN sites that come at the beginning of urllist. It will later check for each module if there is a local copy of the most recent version.

Another peculiarity of urllist is that the site that we could successfully fetch the last file from automatically gets a preference token and is tried as the first site for the next request. So if you add a new site at runtime it may happen that the previously preferred site will be tried another time. This means that if you want to disallow a site for the next transfer, it must be explicitly removed from urllist.

SECURITY

There's no strong security layer in CPAN.pm. CPAN.pm helps you to install foreign, unmasked, unsigned code on your machine. We compare to a checksum that comes from the net just as the distribution file itself. If somebody has managed to tamper with the distribution file, they may have as well tampered with the CHECKSUMS file. Future development will go towards strong authentication.

EXPORT

Most functions in package CPAN are exported per default. The reason for this is that the primary use is intended for the cpan shell or for one-liners.

POPULATE AN INSTALLATION WITH LOTS OF MODULES

Populating a freshly installed perl with my favorite modules is pretty easy if you maintain a private bundle definition file. To get a useful blueprint of a bundle definition file, the command autobundle can be used on the CPAN shell command line. This command writes a bundle definition file for all modules that are installed for the currently running perl interpreter. It's recommended to run this command only once and from then on maintain the file manually under a private name, say Bundle/my_bundle.pm. With a clever bundle file you can then simply say

```
cpan> install Bundle::my_bundle
```

then answer a few questions and then go out for a coffee.

Maintaining a bundle definition file means keeping track of two things: dependencies and interactivity. CPAN.pm sometimes fails on calculating dependencies because not all modules define all MakeMaker attributes correctly, so a bundle definition file should specify prerequisites as early as possible. On the other hand, it's a bit annoying that many distributions need some interactive configuring. So what I try to accomplish in my private bundle file is to have the packages that need to be configured early in the file and the gentle ones later, so I can go out after a few minutes and leave CPAN.pm untended.

WORKING WITH CPAN.pm BEHIND FIREWALLS

Thanks to Graham Barr for contributing the following paragraphs about the interaction between perl, and various firewall configurations. For further informations on firewalls, it is recommended to consult the documentation that comes with the ncftp program. If you are unable to go through the firewall with a simple Perl setup, it is very likely that you can configure ncftp so that it works for your firewall.

Three basic types of firewalls

Firewalls can be categorized into three basic types.

http firewall

This is where the firewall machine runs a web server and to access the outside world you must do it via the web server. If you set environment variables like `http_proxy` or `ftp_proxy` to a values beginning with `http://` or in your web browser you have to set proxy information then you know you are running an http firewall.

To access servers outside these types of firewalls with perl (even for ftp) you will need to use LWP.

ftp firewall

This where the firewall machine runs an ftp server. This kind of firewall will only let you access ftp servers outside the firewall. This is usually done by connecting to the firewall with ftp, then entering a username like "user@outside.host.com"

To access servers outside these type of firewalls with perl you will need to use Net::FTP.

One way visibility

I say one way visibility as these firewalls try to make themselves look invisible to the users inside the firewall. An FTP data connection is normally created by sending the remote server your IP address and then listening for the connection. But the remote server will not be able to connect to you because of the firewall. So for these types of firewall FTP connections need to be done in a passive mode.

There are two that I can think off.

SOCKS

If you are using a SOCKS firewall you will need to compile perl and link it with the SOCKS library, this is what is normally called a 'socksified' perl. With this executable you will be able to connect to servers outside the firewall as if it is not there.

IP Masquerade

This is the firewall implemented in the Linux kernel, it allows you to hide a complete network behind one IP address. With this firewall no special compiling is needed as you can access hosts directly.

Configuring lynx or ncftp for going through a firewall

If you can go through your firewall with e.g. lynx, presumably with a command such as

```
/usr/local/bin/lynx -pscott:tiger
```

then you would configure CPAN.pm with the command

```
o conf lynx "/usr/local/bin/lynx -pscott:tiger"
```

That's all. Similarly for ncftp or ftp, you would configure something like

```
o conf ncftp "/usr/bin/ncftp -f /home/scott/ncftplogin.cfg"
```

Your mileage may vary...

FAQ

- 1) I installed a new version of module X but CPAN keeps saying, I have the old version installed

Most probably you **do** have the old version installed. This can happen if a module installs itself into a different directory in the @INC path than it was previously installed. This is not really a CPAN.pm problem, you would have the same problem when installing the module manually. The easiest way to prevent this behaviour is to add the argument UNINST=1 to the make install call, and that is why many people add this argument permanently by configuring

```
o conf make_install_arg UNINST=1
```

- 2) So why is UNINST=1 not the default?

Because there are people who have their precise expectations about who may install where in the @INC path and who uses which @INC array. In fine tuned environments UNINST=1 can cause damage.

- 3) I want to clean up my mess, and install a new perl along with all modules I have. How do I go about it?

Run the autobundle command for your old perl and optionally rename the resulting bundle file (e.g. Bundle/mybundle.pm), install the new perl with the Configure option prefix, e.g.

```
./Configure -Dprefix=/usr/local/perl-5.6.78.9
```

Install the bundle file you produced in the first step with something like

```
cpan> install Bundle::mybundle
```

and you're done.

- 4) When I install bundles or multiple modules with one command there is too much output to keep track of.

You may want to configure something like

```
o conf make_arg "| tee -ai /root/.cpan/logs/make.out"
o conf make_install_arg "| tee -ai /root/.cpan/logs/make_install.out"
```

so that STDOUT is captured in a file for later inspection.

- 5) I am not root, how can I install a module in a personal directory?

You will most probably like something like this:

```
o conf makepl_arg "LIB=~/.myperl/lib \
                  INSTALLMAN1DIR=~/.myperl/man/man1 \
                  INSTALLMAN3DIR=~/.myperl/man/man3"
install Sybase::Sybperl
```

You can make this setting permanent like all o conf settings with o conf commit.

You will have to add ~/.myperl/man to the MANPATH environment variable and also tell your perl programs to look into ~/.myperl/lib, e.g. by including

```
use lib "$ENV{HOME}/myperl/lib";
```

or setting the PERL5LIB environment variable.

Another thing you should bear in mind is that the UNINST parameter should never be set if you are not root.

- 6) How to get a package, unwrap it, and make a change before building it?

```
look Sybase::Sybperl
```

- 7) I installed a Bundle and had a couple of fails. When I retried, everything resolved nicely. Can this be fixed to work on first try?

The reason for this is that CPAN does not know the dependencies of all modules when it starts out. To decide about the additional items to install, it just uses data found in the generated Makefile. An undetected missing piece breaks the process. But it may well be that your Bundle installs some prerequisite later than some depending item and thus your second try is able to resolve everything. Please note, CPAN.pm does not know the dependency tree in advance and cannot sort the queue of things to install in a topologically correct order. It resolves perfectly well IFF all modules declare the prerequisites correctly with the PREREQ_PM attribute to MakeMaker. For bundles which fail and you need to install often, it is recommended sort the Bundle definition file manually. It is planned to improve the metadata situation for dependencies on CPAN in general, but this will still take some time.

- 8) In our intranet we have many modules for internal use. How can I integrate these modules with CPAN.pm but without uploading the modules to CPAN?

Have a look at the CPAN::Site module.

- 9) When I run CPAN's shell, I get error msg about line 1 to 4, setting meta input/output via the /etc/inputrc file.

Some versions of readline are picky about capitalization in the /etc/inputrc file and specifically RedHat 6.2 comes with a /etc/inputrc that contains the word `on` in lowercase. Change the occurrences of `on` to `On` and the bug should disappear.

- 10) Some authors have strange characters in their names.

Internally CPAN.pm uses the UTF-8 charset. If your terminal is expecting ISO-8859-1 charset, a converter can be activated by setting `term_is_latin` to a true value in your config file. One way of doing so would be

```
cpan> ! $CPAN::Config->{term_is_latin}=1
```

Extended support for converters will be made available as soon as perl becomes stable with regard to charset issues.

BUGS

We should give coverage for **all** of the CPAN and not just the PAUSE part, right? In this discussion CPAN and PAUSE have become equal — but they are not. PAUSE is `authors/`, `modules/` and `scripts/`. CPAN is PAUSE plus the `clpa/`, `doc/`, `misc/`, `ports/`, and `src/`.

Future development should be directed towards a better integration of the other parts.

If a Makefile.PL requires special customization of libraries, prompts the user for special input, etc. then you may find CPAN is not able to build the distribution. In that case, you should attempt the traditional method of building a Perl module package from a shell.

AUTHOR

Andreas Koenig <andreas.koenig@anima.de>

TRANSLATIONS

Kawai,Takanori provides a Japanese translation of this manpage at <http://member.nifty.ne.jp/hippo2000/perltops/CPAN.htm>

SEE ALSO

perl(1), CPAN::Nox(3)

NAME

CPAN::FirstTime – Utility for CPAN::Config file Initialization

SYNOPSIS

```
CPAN::FirstTime::init()
```

DESCRIPTION

The init routine asks a few questions and writes a CPAN::Config file. Nothing special.

NAME

CPAN::Nox – Wrapper around CPAN.pm without using any XS module

SYNOPSIS

Interactive mode:

```
perl -MCPAN::Nox -e shell;
```

DESCRIPTION

This package has the same functionality as CPAN.pm, but tries to prevent the usage of compiled extensions during its own execution. Its primary purpose is a rescue in case you upgraded perl and broke binary compatibility somehow.

SEE ALSO

CPAN(3)

NAME

Cwd – get pathname of current working directory

SYNOPSIS

```
use Cwd;
my $dir = getcwd;

use Cwd 'abs_path';
my $abs_path = abs_path($file);
```

DESCRIPTION

This module provides functions for determining the pathname of the current working directory. It is recommended that `getcwd` (or another `*cwd()` function) be used in *all* code to ensure portability.

By default, it exports the functions `cwd()`, `getcwd()`, `fastcwd()`, and `fastgetcwd()` into the caller's namespace.

getcwd and friends

Each of these functions are called without arguments and return the absolute path of the current working directory.

getcwd

```
my $cwd = getcwd();
```

Returns the current working directory.

Re-implements the `getcwd(3)` (or `getwd(3)`) functions in Perl.

Taint-safe.

cwd

```
my $cwd = cwd();
```

The `cwd()` is the most natural form for the current architecture. For most systems it is identical to `'pwd'` (but without the trailing line terminator).

Taint-safe.

fastcwd

```
my $cwd = fastcwd();
```

A more dangerous version of `getcwd()`, but potentially faster.

It might conceivably `chdir()` you out of a directory that it can't `chdir()` you back into. If `fastcwd` encounters a problem it will return `undef` but will probably leave you in a different directory. For a measure of extra security, if everything appears to have worked, the `fastcwd()` function will check that it leaves you in the same directory that it started in. If it has changed it will `die` with the message "Unstable directory path, current directory changed unexpectedly". That should never happen.

fastgetcwd

```
my $cwd = fastgetcwd();
```

The `fastgetcwd()` function is provided as a synonym for `cwd()`.

abs_path and friends

These functions are exported only on request. They each take a single argument and return the absolute pathname for it.

abs_path

```
my $abs_path = abs_path($file);
```

Uses the same algorithm as `getcwd()`. Symbolic links and relative-path components (`."` and `".."`)

are resolved to return the canonical pathname, just like `realpath(3)`.

Taint-safe.

`realpath`

```
my $abs_path = realpath($file);
```

A synonym for `abs_path()`.

Taint-safe.

`fast_abs_path`

```
my $abs_path = fast_abs_path($file);
```

A more dangerous, but potentially faster version of `abs_path`.

This function is **Not** taint-safe : you can't use it in programs that work under taint mode.

`$ENV{PWD}`

If you ask to override your `chdir()` built-in function,

```
use Cwd qw(chdir);
```

then your `PWD` environment variable will be kept up to date. Note that it will only be kept up to date if all packages which use `chdir` import it from `Cwd`.

NOTES

- Since the path separators are different on some operating systems ('/' on Unix, ':' on MacPerl, etc...) we recommend you use the `File::Spec` modules wherever portability is a concern.
- Actually, on Mac OS, the `getcwd()`, `fastgetcwd()` and `fastcwd()` functions are all aliases for the `cwd()` function, which, on Mac OS, calls 'pwd'. Likewise, the `abs_path()` function is an alias for `fast_abs_path()`.

SEE ALSO

[File::chdir](#)

NAME

Data::Dumper – stringified perl data structures, suitable for both printing and eval

SYNOPSIS

```
use Data::Dumper;

# simple procedural interface
print Dumper($foo, $bar);

# extended usage with names
print Data::Dumper->Dump([$foo, $bar], [qw(foo *ary)]);

# configuration variables
{
    local $Data::Dump::Purity = 1;
    eval Data::Dumper->Dump([$foo, $bar], [qw(foo *ary)]);
}

# OO usage
$d = Data::Dumper->new([$foo, $bar], [qw(foo *ary)]);
...
print $d->Dump;
...
$d->Purity(1)->Terse(1)->Deepcopy(1);
eval $d->Dump;
```

DESCRIPTION

Given a list of scalars or reference variables, writes out their contents in perl syntax. The references can also be objects. The contents of each variable is output in a single Perl statement. Handles self-referential structures correctly.

The return value can be `eval`d to get back an identical copy of the original reference structure.

Any references that are the same as one of those passed in will be named `$VARn` (where *n* is a numeric suffix), and other duplicate references to substructures within `$VARn` will be appropriately labeled using arrow notation. You can specify names for individual values to be dumped if you use the `Dump()` method, or you can change the default `$VAR` prefix to something else. See `$Data::Dumper::Varname` and `$Data::Dumper::Terse` below.

The default output of self-referential structures can be `eval`d, but the nested references to `$VARn` will be undefined, since a recursive structure cannot be constructed using one Perl statement. You should set the `Purity` flag to 1 to get additional statements that will correctly fill in these references.

In the extended usage form, the references to be dumped can be given user-specified names. If a name begins with a `*`, the output will describe the dereferenced type of the supplied reference for hashes and arrays, and coderefs. Output of names will be avoided where possible if the `Terse` flag is set.

In many cases, methods that are used to set the internal state of the object will return the object itself, so method calls can be conveniently chained together.

Several styles of output are possible, all controlled by setting the `Indent` flag. See [Configuration Variables or Methods](#) below for details.

Methods

PACKAGE-new(*ARRAYREF* [, *ARRAYREF*])

Returns a newly created `Data::Dumper` object. The first argument is an anonymous array of values to be dumped. The optional second argument is an anonymous array of names for the values. The names need not have a leading `$` sign, and must be comprised of alphanumeric characters. You can begin a name with a `*` to specify that the dereferenced type must be dumped instead of the reference

itself, for ARRAY and HASH references.

The prefix specified by `$Data::Dumper::Varname` will be used with a numeric suffix if the name for a value is undefined.

`Data::Dumper` will catalog all references encountered while dumping the values. Cross-references (in the form of names of substructures in perl syntax) will be inserted at all possible points, preserving any structural interdependencies in the original set of values. Structure traversal is depth-first, and proceeds in order from the first supplied value to the last.

`$OBJ-Dump` or `PACKAGE-Dump(ARRAYREF [, ARRAYREF])`

Returns the stringified form of the values stored in the object (preserving the order in which they were supplied to `new`), subject to the configuration options below. In a list context, it returns a list of strings corresponding to the supplied values.

The second form, for convenience, simply calls the `new` method on its arguments before dumping the object immediately.

`$OBJ-Seen([HASHREF])`

Queries or adds to the internal table of already encountered references. You must use `Reset` to explicitly clear the table if needed. Such references are not dumped; instead, their names are inserted wherever they are encountered subsequently. This is useful especially for properly dumping subroutine references.

Expects an anonymous hash of name = value pairs. Same rules apply for names as in `new`. If no argument is supplied, will return the "seen" list of name = value pairs, in a list context. Otherwise, returns the object itself.

`$OBJ-Values([ARRAYREF])`

Queries or replaces the internal array of values that will be dumped. When called without arguments, returns the values. Otherwise, returns the object itself.

`$OBJ-Names([ARRAYREF])`

Queries or replaces the internal array of user supplied names for the values that will be dumped. When called without arguments, returns the names. Otherwise, returns the object itself.

`$OBJ-Reset`

Clears the internal table of "seen" references and returns the object itself.

Functions

`Dumper(LIST)`

Returns the stringified form of the values in the list, subject to the configuration options below. The values will be named `$VARn` in the output, where *n* is a numeric suffix. Will return a list of strings in a list context.

Configuration Variables or Methods

Several configuration variables can be used to control the kind of output generated when using the procedural interface. These variables are usually `localized` in a block so that other parts of the code are not affected by the change.

These variables determine the default state of the object created by calling the `new` method, but cannot be used to alter the state of the object thereafter. The equivalent method names should be used instead to query or set the internal state of the object.

The method forms return the object itself when called with arguments, so that they can be chained together nicely.

`$Data::Dumper::Indent` or `$OBJ-Indent([NEWVAL])`

Controls the style of indentation. It can be set to 0, 1, 2 or 3. Style 0 spews output without any newlines, indentation, or spaces between list items. It is the most compact format possible that can still be called valid perl. Style 1 outputs a readable form with newlines but no fancy indentation (each level in the structure is simply indented by a fixed amount of whitespace). Style 2 (the default) outputs a very readable form which takes into account the length of hash keys (so the hash value lines up). Style 3 is like style 2, but also annotates the elements of arrays with their index (but the comment is on its own line, so array output consumes twice the number of lines). Style 2 is the default.

`$Data::Dumper::Purity` or `$OBJ-Purity([NEWVAL])`

Controls the degree to which the output can be eval'd to recreate the supplied reference structures. Setting it to 1 will output additional perl statements that will correctly recreate nested references. The default is 0.

`$Data::Dumper::Pad` or `$OBJ-Pad([NEWVAL])`

Specifies the string that will be prefixed to every line of the output. Empty string by default.

`$Data::Dumper::Varname` or `$OBJ-Varname([NEWVAL])`

Contains the prefix to use for tagging variable names in the output. The default is "VAR".

`$Data::Dumper::Useqq` or `$OBJ-Useqq([NEWVAL])`

When set, enables the use of double quotes for representing string values. Whitespace other than space will be represented as `[\n\t\r]`, "unsafe" characters will be backslashed, and unprintable characters will be output as quoted octal integers. Since setting this variable imposes a performance penalty, the default is 0. `Dump()` will run slower if this flag is set, since the fast XSUB implementation doesn't support it yet.

`$Data::Dumper::Terse` or `$OBJ-Terse([NEWVAL])`

When set, `Data::Dumper` will emit single, non-self-referential values as atoms/terms rather than statements. This means that the `$VARn` names will be avoided where possible, but be advised that such output may not always be parseable by `eval`.

`$Data::Dumper::Freezer` or `$OBJ-Freezer([NEWVAL])`

Can be set to a method name, or to an empty string to disable the feature. `Data::Dumper` will invoke that method via the object before attempting to stringify it. This method can alter the contents of the object (if, for instance, it contains data allocated from C), and even rebless it in a different package. The client is responsible for making sure the specified method can be called via the object, and that the object ends up containing only perl data types after the method has been called. Defaults to an empty string.

`$Data::Dumper::Toaster` or `$OBJ-Toaster([NEWVAL])`

Can be set to a method name, or to an empty string to disable the feature. `Data::Dumper` will emit a method call for any objects that are to be dumped using the syntax `bless(DATA, CLASS)->METHOD()`. Note that this means that the method specified will have to perform any modifications required on the object (like creating new state within it, and/or reblessing it in a different package) and then return it. The client is responsible for making sure the method can be called via the object, and that it returns a valid object. Defaults to an empty string.

`$Data::Dumper::Deepcopy` or `$OBJ-Deepcopy([NEWVAL])`

Can be set to a boolean value to enable deep copies of structures. Cross-referencing will then only be done when absolutely essential (i.e., to break reference cycles). Default is 0.

`$Data::Dumper::Quotekeys` or `$OBJ-Quotekeys([NEWVAL])`

Can be set to a boolean value to control whether hash keys are quoted. A false value will avoid quoting hash keys when it looks like a simple string. Default is 1, which will always enclose hash keys in quotes.

`$Data::Dumper::Bless` or `$OBJ-Bless([NEWVAL])`

Can be set to a string that specifies an alternative to the `bless` builtin operator used to create objects. A function with the specified name should exist, and should accept the same arguments as the builtin. Default is `bless`.

`$Data::Dumper::Maxdepth` or `$OBJ-Maxdepth([NEWVAL])`

Can be set to a positive integer that specifies the depth beyond which we don't venture into a structure. Has no effect when `Data::Dumper::Purity` is set. (Useful in debugger when we often don't want to see more than enough). Default is 0, which means there is no maximum depth.

`$Data::Dumper::Useperl` or `$OBJ-Useperl([NEWVAL])`

Can be set to a boolean value which controls whether the pure Perl implementation of `Data::Dumper` is used. The `Data::Dumper` module is a dual implementation, with almost all functionality written in both pure Perl and also in XS ('C'). Since the XS version is much faster, it will always be used if possible. This option lets you override the default behavior, usually for testing purposes only. Default is 0, which means the XS implementation will be used if possible.

`$Data::Dumper::Sortkeys` or `$OBJ-Sortkeys([NEWVAL])`

Can be set to a boolean value to control whether hash keys are dumped in sorted order. A true value will cause the keys of all hashes to be dumped in Perl's default sort order. Can also be set to a subroutine reference which will be called for each hash that is dumped. In this case `Data::Dumper` will call the subroutine once for each hash, passing it the reference of the hash. The purpose of the subroutine is to return a reference to an array of the keys that will be dumped, in the order that they should be dumped. Using this feature, you can control both the order of the keys, and which keys are actually used. In other words, this subroutine acts as a filter by which you can exclude certain keys from being dumped. Default is 0, which means that hash keys are not sorted.

`$Data::Dumper::Deparse` or `$OBJ-Deparse([NEWVAL])`

Can be set to a boolean value to control whether code references are turned into perl source code. If set to a true value, `B::Deparse` will be used to get the source of the code reference. Using this option will force using the Perl implementation of the dumper, since the fast XSUB implementation doesn't support it.

Caution : use this option only if you know that your coderefs will be properly reconstructed by `B::Deparse`.

Exports

Dumper

EXAMPLES

Run these code snippets to get a quick feel for the behavior of this module. When you are through with these examples, you may want to add or change the various configuration variables described above, to see their behavior. (See the testsuite in the `Data::Dumper` distribution for more examples.)

```
use Data::Dumper;

package Foo;
sub new {bless {'a' => 1, 'b' => sub { return "foo" }}, $_[0]};

package Fuz;
# a weird REF-REF-SCALAR object
sub new {bless \($_ = \ 'fu\'z'), $_[0]};

package main;
$foo = Foo->new;
$fuz = Fuz->new;
$boo = [ 1, [], "abcd", \*foo,
         {1 => 'a', 023 => 'b', 0x45 => 'c'},
         \\ "p\q\'r", $foo, $fuz];
```

```

#####
# simple usage
#####

$bar = eval(Dumper($boo));
print($@) if $@;
print Dumper($boo), Dumper($bar); # pretty print (no array indices)

$Data::Dumper::Terse = 1;          # don't output names where feasible
$Data::Dumper::Indent = 0;        # turn off all pretty print
print Dumper($boo), "\n";

$Data::Dumper::Indent = 1;        # mild pretty print
print Dumper($boo);

$Data::Dumper::Indent = 3;        # pretty print with array indices
print Dumper($boo);

$Data::Dumper::Useqq = 1;         # print strings in double quotes
print Dumper($boo);

#####
# recursive structures
#####

@c = ('c');
$c = \@c;
$b = {};
$a = [1, $b, $c];
$b->{a} = $a;
$b->{b} = $a->[1];
$b->{c} = $a->[2];
print Data::Dumper->Dump([$a,$b,$c], [qw(a b c)]);

$Data::Dumper::Purity = 1;        # fill in the holes for eval
print Data::Dumper->Dump([$a, $b], [qw(*a b)]); # print as @a
print Data::Dumper->Dump([$b, $a], [qw(*b a)]); # print as %b

$Data::Dumper::Deepcopy = 1;      # avoid cross-refs
print Data::Dumper->Dump([$b, $a], [qw(*b a)]);

$Data::Dumper::Purity = 0;        # avoid cross-refs
print Data::Dumper->Dump([$b, $a], [qw(*b a)]);

#####
# deep structures
#####

$a = "pearl";
$b = [ $a ];
$c = { 'b' => $b };
$d = [ $c ];
$e = { 'd' => $d };
$f = { 'e' => $e };
print Data::Dumper->Dump([$f], [qw(f)]);

$Data::Dumper::Maxdepth = 3;      # no deeper than 3 refs down
print Data::Dumper->Dump([$f], [qw(f)]);

#####
# object-oriented usage

```

```

#####
$d = Data::Dumper->new([$a,$b], [qw(a b)]);
$d->Seen({'*c' => $c});          # stash a ref without printing it
$d->Indent(3);
print $d->Dump;
$d->Reset->Purity(0);           # empty the seen cache
print join "----\n", $d->Dump;

#####
# persistence
#####

package Foo;
sub new { bless { state => 'awake' }, shift }
sub Freeze {
    my $s = shift;
    print STDERR "preparing to sleep\n";
    $s->{state} = 'asleep';
    return bless $s, 'Foo::ZZZ';
}

package Foo::ZZZ;
sub Thaw {
    my $s = shift;
    print STDERR "waking up\n";
    $s->{state} = 'awake';
    return bless $s, 'Foo';
}

package Foo;
use Data::Dumper;
$a = Foo->new;
$b = Data::Dumper->new([$a], ['c']);
$b->Freezer('Freeze');
$b->Toaster('Thaw');
$c = $b->Dump;
print $c;
$d = eval $c;
print Data::Dumper->Dump([$d], ['d']);

#####
# symbol substitution (useful for recreating CODE refs)
#####

sub foo { print "foo speaking\n" }
*other = \&foo;
$bar = [ \&other ];
$d = Data::Dumper->new([\&other,$bar],['*other','bar']);
$d->Seen({'*foo' => \&foo });
print $d->Dump;

#####
# sorting and filtering hash keys
#####

$Data::Dumper::Sortkeys = \&my_filter;
my $foo = { map { (ord, "$_$_$_") } 'I'..'Q' };
my $bar = { %$foo };

```

```

my $baz = { reverse %$foo };
print Dumper [ $foo, $bar, $baz ];

sub my_filter {
    my ($hash) = @_;
    # return an array ref containing the hash keys to dump
    # in the order that you want them to be dumped
    return [
        # Sort the keys of %$foo in reverse numeric order
        $hash eq $foo ? (sort {$b <=> $a} keys %$hash) :
        # Only dump the odd number keys of %$bar
        $hash eq $bar ? (grep {$_ % 2} keys %$hash) :
        # Sort keys in default order for all other hashes
        (sort keys %$hash)
    ];
}

```

BUGS

Due to limitations of Perl subroutine call semantics, you cannot pass an array or hash. Prepend it with a \ to pass its reference instead. This will be remedied in time, now that Perl has subroutine prototypes. For now, you need to use the extended usage form, and prepend the name with a * to output it as a hash or array.

`Data::Dumper` cheats with CODE references. If a code reference is encountered in the structure being processed (and if you haven't set the `Deparse` flag), an anonymous subroutine that contains the string "DUMMY" will be inserted in its place, and a warning will be printed if `Purity` is set. You can `eval` the result, but bear in mind that the anonymous sub that gets created is just a placeholder. Someday, perl will have a switch to cache-on-demand the string representation of a compiled piece of code, I hope. If you have prior knowledge of all the code refs that your data structures are likely to have, you can use the `Seen` method to pre-seed the internal reference table and make the dumped output point to them, instead. See [EXAMPLES](#) above.

The `Useqq` and `Deparse` flags makes `Dump()` run slower, since the XSUB implementation does not support them.

SCALAR objects have the weirdest looking `bless` workaround.

AUTHOR

Gurusamy Sarathy gsar@activestate.com

Copyright (c) 1996–98 Gurusamy Sarathy. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

VERSION

Version 2.12 (unreleased)

SEE ALSO

`perl(1)`

NAME

DB – programmatic interface to the Perl debugging API (draft, subject to change)

SYNOPSIS

```
package CLIENT;
use DB;
@ISA = qw(DB);

# these (inherited) methods can be called by the client

CLIENT->register()      # register a client package name
CLIENT->done()          # de-register from the debugging API
CLIENT->skippkg('hide::hide') # ask DB not to stop in this package
CLIENT->cont([WHERE])   # run some more (until BREAK or another breakpt)
CLIENT->step()          # single step
CLIENT->next()          # step over
CLIENT->ret()           # return from current subroutine
CLIENT->backtrace()     # return the call stack description
CLIENT->ready()         # call when client setup is done
CLIENT->trace_toggle()  # toggle subroutine call trace mode
CLIENT->subs([SUBS])    # return subroutine information
CLIENT->files()         # return list of all files known to DB
CLIENT->lines()         # return lines in currently loaded file
CLIENT->loadfile(FILE,LINE) # load a file and let other clients know
CLIENT->lineevents()    # return info on lines with actions
CLIENT->set_break([WHERE],[COND])
CLIENT->set_tbreak([WHERE])
CLIENT->clr_breaks([LIST])
CLIENT->set_action(WHERE,ACTION)
CLIENT->clr_actions([LIST])
CLIENT->evalcode(String) # eval STRING in executing code's context
CLIENT->prestop([String]) # execute in code context before stopping
CLIENT->poststop([String]) # execute in code context before resuming

# These methods will be called at the appropriate times.
# Stub versions provided do nothing.
# None of these can block.

CLIENT->init()          # called when debug API inits itself
CLIENT->stop(FILE,LINE) # when execution stops
CLIENT->idle()          # while stopped (can be a client event loop)
CLIENT->cleanup()       # just before exit
CLIENT->output(LIST)    # called to print any output that API must show
```

DESCRIPTION

Perl debug information is frequently required not just by debuggers, but also by modules that need some "special" information to do their job properly, like profilers.

This module abstracts and provides all of the hooks into Perl internal debugging functionality, so that various implementations of Perl debuggers (or packages that want to simply get at the "privileged" debugging data) can all benefit from the development of this common code. Currently used by Swat, the perl/Tk GUI debugger.

Note that multiple "front-ends" can latch into this debugging API simultaneously. This is intended to facilitate things like debugging with a command line and GUI at the same time, debugging debuggers etc. [Sounds nice, but this needs some serious support — GSAR]

In particular, this API does **not** provide the following functions:

- data display
- command processing
- command alias management
- user interface (tty or graphical)

These are intended to be services performed by the clients of this API.

This module attempts to be squeaky clean w.r.t `use strict`; and when warnings are enabled.

Global Variables

The following "public" global names can be read by clients of this API. Beware that these should be considered "readonly".

`$DB::sub`

Name of current executing subroutine.

`%DB::sub`

The keys of this hash are the names of all the known subroutines. Each value is an encoded string that has the `sprintf(3)` format `("%s:%d-%d", filename, fromline, toline)`.

`$DB::single`

Single-step flag. Will be true if the API will stop at the next statement.

`$DB::signal`

Signal flag. Will be set to a true value if a signal was caught. Clients may check for this flag to abort time-consuming operations.

`$DB::trace`

This flag is set to true if the API is tracing through subroutine calls.

`@DB::args`

Contains the arguments of current subroutine, or the `@ARGV` array if in the `toplevel` context.

`@DB::dbline`

List of lines in currently loaded file.

`%DB::dbline`

Actions in current file (keys are line numbers). The values are strings that have the `sprintf(3)` format `("%s\000%s", breakcondition, actioncode)`.

`$DB::package`

Package namespace of currently executing code.

`$DB::filename`

Currently loaded filename.

`$DB::subname`

Fully qualified name of currently executing subroutine.

`$DB::lineno`

Line number that will be executed next.

API Methods

The following are methods in the DB base class. A client must access these methods by inheritance (**not** by calling them directly), since the API keeps track of clients through the inheritance mechanism.

CLIENT-register()

register a client object/package

CLIENT-evalcode(String)

eval String in executing code context

CLIENT-skippkg('D::hide')

ask DB not to stop in these packages

CLIENT-run()

run some more (until a breakpoint is reached)

CLIENT-step()

single step

CLIENT-next()

step over

CLIENT-done()

de-register from the debugging API

Client Callback Methods

The following "virtual" methods can be defined by the client. They will be called by the API at appropriate points. Note that unless specified otherwise, the debug API only defines empty, non-functional default versions of these methods.

CLIENT-init()

Called after debug API inits itself.

CLIENT-prestop([String])

Usually inherited from DB package. If no arguments are passed, returns the prestop action string.

CLIENT-stop()

Called when execution stops (w/ args file, line).

CLIENT-idle()

Called while stopped (can be a client event loop).

CLIENT-poststop([String])

Usually inherited from DB package. If no arguments are passed, returns the poststop action string.

CLIENT-evalcode(String)

Usually inherited from DB package. Ask for a String to be eval-ed in executing code context.

CLIENT-cleanup()

Called just before exit.

CLIENT-output(List)

Called when API must show a message (warnings, errors etc.).

BUGS

The interface defined by this module is missing some of the later additions to perl's debugging functionality. As such, this interface should be considered highly experimental and subject to change.

AUTHOR

Gurusamy Sarathy gsar@activestate.com

This code heavily adapted from an early version of perl5db.pl attributable to Larry Wall and the Perl Porters.

NAME

DB_File – Perl5 access to Berkeley DB version 1.x

SYNOPSIS

```
use DB_File;

[$X =] tie %hash, 'DB_File', [$filename, $flags, $mode, $DB_HASH] ;
[$X =] tie %hash, 'DB_File', $filename, $flags, $mode, $DB_BTREE ;
[$X =] tie @array, 'DB_File', $filename, $flags, $mode, $DB_RECNO ;

$status = $X->del($key [, $flags]) ;
$status = $X->put($key, $value [, $flags]) ;
$status = $X->get($key, $value [, $flags]) ;
$status = $X->seq($key, $value, $flags) ;
$status = $X->sync([$flags]) ;
$status = $X->fd ;

# BTREE only
$count = $X->get_dup($key) ;
@list = $X->get_dup($key) ;
%list = $X->get_dup($key, 1) ;
$status = $X->find_dup($key, $value) ;
$status = $X->del_dup($key, $value) ;

# RECNO only
$a = $X->length;
$a = $X->pop ;
$X->push(list);
$a = $X->shift;
$X->unshift(list);
@r = $X->splice(offset, length, elements);

# DBM Filters
$old_filter = $db->filter_store_key ( sub { ... } ) ;
$old_filter = $db->filter_store_value( sub { ... } ) ;
$old_filter = $db->filter_fetch_key ( sub { ... } ) ;
$old_filter = $db->filter_fetch_value( sub { ... } ) ;

untie %hash ;
untie @array ;
```

DESCRIPTION

DB_File is a module which allows Perl programs to make use of the facilities provided by Berkeley DB version 1.x (if you have a newer version of DB, see [Using DB_File with Berkeley DB version 2 or greater](#)). It is assumed that you have a copy of the Berkeley DB manual pages at hand when reading this documentation. The interface defined here mirrors the Berkeley DB interface closely.

Berkeley DB is a C library which provides a consistent interface to a number of database formats. **DB_File** provides an interface to all three of the database types currently supported by Berkeley DB.

The file types are:

DB_HASH

This database type allows arbitrary key/value pairs to be stored in data files. This is equivalent to the functionality provided by other hashing packages like DBM, NDBM, ODBM, GDBM, and SDBM. Remember though, the files created using DB_HASH are not compatible with any of the other packages mentioned.

A default hashing algorithm, which will be adequate for most applications, is built into Berkeley DB.

If you do need to use your own hashing algorithm it is possible to write your own in Perl and have **DB_File** use it instead.

DB_BTREE

The btree format allows arbitrary key/value pairs to be stored in a sorted, balanced binary tree.

As with the DB_HASH format, it is possible to provide a user defined Perl routine to perform the comparison of keys. By default, though, the keys are stored in lexical order.

DB_RECNO

DB_RECNO allows both fixed-length and variable-length flat text files to be manipulated using the same key/value pair interface as in DB_HASH and DB_BTREE. In this case the key will consist of a record (line) number.

Using DB_File with Berkeley DB version 2 or greater

Although **DB_File** is intended to be used with Berkeley DB version 1, it can also be used with version 2, 3 or 4. In this case the interface is limited to the functionality provided by Berkeley DB 1.x. Anywhere the version 2 or greater interface differs, **DB_File** arranges for it to work like version 1. This feature allows **DB_File** scripts that were built with version 1 to be migrated to version 2 or greater without any changes.

If you want to make use of the new features available in Berkeley DB 2.x or greater, use the Perl module **BerkeleyDB** instead.

Note: The database file format has changed multiple times in Berkeley DB version 2, 3 and 4. If you cannot recreate your databases, you must dump any existing databases with either the `db_dump` or the `db_dump185` utility that comes with Berkeley DB. Once you have rebuilt DB_File to use Berkeley DB version 2 or greater, your databases can be recreated using `db_load`. Refer to the Berkeley DB documentation for further details.

Please read "[COPYRIGHT](#)" before using version 2.x or greater of Berkeley DB with DB_File.

Interface to Berkeley DB

DB_File allows access to Berkeley DB files using the `tie()` mechanism in Perl 5 (for full details, see [tie\(\)](#)). This facility allows **DB_File** to access Berkeley DB files using either an associative array (for DB_HASH & DB_BTREE file types) or an ordinary array (for the DB_RECNO file type).

In addition to the `tie()` interface, it is also possible to access most of the functions provided in the Berkeley DB API directly. See [THE API INTERFACE](#).

Opening a Berkeley DB Database File

Berkeley DB uses the function `dbopen()` to open or create a database. Here is the C prototype for `dbopen()`:

```
DB*
dbopen (const char* file, int flags, int mode,
        DBTYPE type, const void * openinfo)
```

The parameter `type` is an enumeration which specifies which of the 3 interface methods (DB_HASH, DB_BTREE or DB_RECNO) is to be used. Depending on which of these is actually chosen, the final parameter, `openinfo` points to a data structure which allows tailoring of the specific interface method.

This interface is handled slightly differently in **DB_File**. Here is an equivalent call using **DB_File**:

```
tie %array, 'DB_File', $filename, $flags, $mode, $DB_HASH ;
```

The `filename`, `flags` and `mode` parameters are the direct equivalent of their `dbopen()` counterparts. The final parameter `$DB_HASH` performs the function of both the `type` and `openinfo` parameters in `dbopen()`.

In the example above `$DB_HASH` is actually a pre-defined reference to a hash object. **DB_File** has three of these pre-defined references. Apart from `$DB_HASH`, there is also `$DB_BTREE` and `$DB_RECNO`.

The keys allowed in each of these pre-defined references is limited to the names used in the equivalent C structure. So, for example, the `$DB_HASH` reference will only allow keys called `bsize`, `cachesize`, `ffactor`, `hash`, `lorder` and `nelem`.

To change one of these elements, just assign to it like this:

```
$DB_HASH->{'cachesize'} = 10000 ;
```

The three predefined variables `$DB_HASH`, `$DB_BTREE` and `$DB_RECNO` are usually adequate for most applications. If you do need to create extra instances of these objects, constructors are available for each file type.

Here are examples of the constructors and the valid options available for `DB_HASH`, `DB_BTREE` and `DB_RECNO` respectively.

```
$a = new DB_File::HASHINFO ;
$a->{'bsize'} ;
$a->{'cachesize'} ;
$a->{'ffactor'} ;
$a->{'hash'} ;
$a->{'lorder'} ;
$a->{'nelem'} ;

$b = new DB_File::BTREEINFO ;
$b->{'flags'} ;
$b->{'cachesize'} ;
$b->{'maxkeypage'} ;
$b->{'minkeypage'} ;
$b->{'psize'} ;
$b->{'compare'} ;
$b->{'prefix'} ;
$b->{'lorder'} ;

$c = new DB_File::RECNOINFO ;
$c->{'bval'} ;
$c->{'cachesize'} ;
$c->{'psize'} ;
$c->{'flags'} ;
$c->{'lorder'} ;
$c->{'reclen'} ;
$c->{'bfname'} ;
```

The values stored in the hashes above are mostly the direct equivalent of their C counterpart. Like their C counterparts, all are set to a default values – that means you don't have to set *all* of the values when you only want to change one. Here is an example:

```
$a = new DB_File::HASHINFO ;
$a->{'cachesize'} = 12345 ;
tie %y, 'DB_File', "filename", $flags, 0777, $a ;
```

A few of the options need extra discussion here. When used, the C equivalent of the keys `hash`, `compare` and `prefix` store pointers to C functions. In **DB_File** these keys are used to store references to Perl subs. Below are templates for each of the subs:

```
sub hash
{
    my ($data) = @_ ;
    ...
    # return the hash value for $data
    return $hash ;
}
```

```

    }

    sub compare
    {
        my ($key, $key2) = @_ ;
        ...
        # return 0 if $key1 eq $key2
        #       -1 if $key1 lt $key2
        #       1 if $key1 gt $key2
        return (-1 , 0 or 1) ;
    }

    sub prefix
    {
        my ($key, $key2) = @_ ;
        ...
        # return number of bytes of $key2 which are
        # necessary to determine that it is greater than $key1
        return $bytes ;
    }

```

See [Changing the BTREE sort order](#) for an example of using the compare template.

If you are using the DB_RECNO interface and you intend making use of bval, you should check out [The 'bval' Option](#).

Default Parameters

It is possible to omit some or all of the final 4 parameters in the call to tie and let them take default values. As DB_HASH is the most common file format used, the call:

```
tie %A, "DB_File", "filename" ;
```

is equivalent to:

```
tie %A, "DB_File", "filename", O_CREAT|O_RDWR, 0666, $DB_HASH ;
```

It is also possible to omit the filename parameter as well, so the call:

```
tie %A, "DB_File" ;
```

is equivalent to:

```
tie %A, "DB_File", undef, O_CREAT|O_RDWR, 0666, $DB_HASH ;
```

See [In Memory Databases](#) for a discussion on the use of undef in place of a filename.

In Memory Databases

Berkeley DB allows the creation of in-memory databases by using NULL (that is, a (char *)0 in C) in place of the filename. **DB_File** uses undef instead of NULL to provide this functionality.

DB_HASH

The DB_HASH file format is probably the most commonly used of the three file formats that **DB_File** supports. It is also very straightforward to use.

A Simple Example

This example shows how to create a database, add key/value pairs to the database, delete keys/value pairs and finally how to enumerate the contents of the database.

```

use warnings ;
use strict ;
use DB_File ;
our (%h, $k, $v) ;

```

```

unlink "fruit" ;
tie %h, "DB_File", "fruit", O_RDWR|O_CREAT, 0666, $DB_HASH
    or die "Cannot open file 'fruit': $!\n";

# Add a few key/value pairs to the file
$h{"apple"} = "red" ;
$h{"orange"} = "orange" ;
$h{"banana"} = "yellow" ;
$h{"tomato"} = "red" ;

# Check for existence of a key
print "Banana Exists\n\n" if $h{"banana"} ;

# Delete a key/value pair.
delete $h{"apple"} ;

# print the contents of the file
while (($k, $v) = each %h)
    { print "$k -> $v\n" }

untie %h ;

```

here is the output:

```

Banana Exists

orange -> orange
tomato -> red
banana -> yellow

```

Note that the like ordinary associative arrays, the order of the keys retrieved is in an apparently random order.

DB_BTREE

The DB_BTREE format is useful when you want to store data in a given order. By default the keys will be stored in lexical order, but as you will see from the example shown in the next section, it is very easy to define your own sorting function.

Changing the BTREE sort order

This script shows how to override the default sorting algorithm that BTREE uses. Instead of using the normal lexical ordering, a case insensitive compare function will be used.

```

use warnings ;
use strict ;
use DB_File ;

my %h ;

sub Compare
{
    my ($key1, $key2) = @_ ;
    "\L$key1" cmp "\L$key2" ;
}

# specify the Perl sub that will do the comparison
$db_BTREE->{'compare'} = \&Compare ;

unlink "tree" ;
tie %h, "DB_File", "tree", O_RDWR|O_CREAT, 0666, $DB_BTREE
    or die "Cannot open file 'tree': $!\n" ;

# Add a key/value pair to the file
$h{'Wall'} = 'Larry' ;

```

```

$h{'Smith'} = 'John' ;
$h{'mouse'} = 'mickey' ;
$h{'duck'} = 'donald' ;

# Delete
delete $h{"duck"} ;

# Cycle through the keys printing them in order.
# Note it is not necessary to sort the keys as
# the btree will have kept them in order automatically.
foreach (keys %h)
    { print "$_\n" }

untie %h ;

```

Here is the output from the code above.

```

mouse
Smith
Wall

```

There are a few point to bear in mind if you want to change the ordering in a BTREE database:

1. The new compare function must be specified when you create the database.
2. You cannot change the ordering once the database has been created. Thus you must use the same compare function every time you access the database.
3. Duplicate keys are entirely defined by the comparison function. In the case-insensitive example above, the keys: 'KEY' and 'key' would be considered duplicates, and assigning to the second one would overwrite the first. If duplicates are allowed for (with the R_DUPS flag discussed below), only a single copy of duplicate keys is stored in the database — so (again with example above) assigning three values to the keys: 'KEY', 'Key', and 'key' would leave just the first key: 'KEY' in the database with three values. For some situations this results in information loss, so care should be taken to provide fully qualified comparison functions when necessary. For example, the above comparison routine could be modified to additionally compare case-sensitively if two keys are equal in the case insensitive comparison:

```

sub compare {
    my($key1, $key2) = @_;
    lc $key1 cmp lc $key2 ||
    $key1 cmp $key2;
}

```

And now you will only have duplicates when the keys themselves are truly the same. (note: in versions of the db library prior to about November 1996, such duplicate keys were retained so it was possible to recover the original keys in sets of keys that compared as equal).

Handling Duplicate Keys

The BTREE file type optionally allows a single key to be associated with an arbitrary number of values. This option is enabled by setting the flags element of \$DB_BTREE to R_DUP when creating the database.

There are some difficulties in using the tied hash interface if you want to manipulate a BTREE database with duplicate keys. Consider this code:

```

use warnings ;
use strict ;
use DB_File ;

my ($filename, %h) ;

$filename = "tree" ;

```

```

unlink $filename ;

# Enable duplicate records
$DB_BTREE->{'flags'} = R_DUP ;

tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
    or die "Cannot open $filename: $!\n";

# Add some key/value pairs to the file
$h{'Wall'} = 'Larry' ;
$h{'Wall'} = 'Brick' ; # Note the duplicate key
$h{'Wall'} = 'Brick' ; # Note the duplicate key and value
$h{'Smith'} = 'John' ;
$h{'mouse'} = 'mickey' ;

# iterate through the associative array
# and print each key/value pair.
foreach (sort keys %h)
    { print "$_ -> $h{$_}\n" }

untie %h ;

```

Here is the output:

```

Smith    -> John
Wall     -> Larry
Wall     -> Larry
Wall     -> Larry
mouse    -> mickey

```

As you can see 3 records have been successfully created with key Wall – the only thing is, when they are retrieved from the database they *seem* to have the same value, namely Larry. The problem is caused by the way that the associative array interface works. Basically, when the associative array interface is used to fetch the value associated with a given key, it will only ever retrieve the first value.

Although it may not be immediately obvious from the code above, the associative array interface can be used to write values with duplicate keys, but it cannot be used to read them back from the database.

The way to get around this problem is to use the Berkeley DB API method called `seq`. This method allows sequential access to key/value pairs. See [THE API INTERFACE](#) for details of both the `seq` method and the API in general.

Here is the script above rewritten using the `seq` API method.

```

use warnings ;
use strict ;
use DB_File ;

my ($filename, $x, %h, $status, $key, $value) ;

$filename = "tree" ;
unlink $filename ;

# Enable duplicate records
$DB_BTREE->{'flags'} = R_DUP ;

$x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
    or die "Cannot open $filename: $!\n";

# Add some key/value pairs to the file
$h{'Wall'} = 'Larry' ;
$h{'Wall'} = 'Brick' ; # Note the duplicate key
$h{'Wall'} = 'Brick' ; # Note the duplicate key and value

```



```

$h{'Smith'} = 'John' ;
$h{'mouse'} = 'mickey' ;

# iterate through the btree using seq
# and print each key/value pair.
$key = $value = 0 ;
for ($status = $x->seq($key, $value, R_FIRST) ;
    $status == 0 ;
    $status = $x->seq($key, $value, R_NEXT) )
{
    print "$key -> $value\n" ;
}

undef $x ;
untie %h ;

```

that prints:

```

Smith    -> John
Wall     -> Brick
Wall     -> Brick
Wall     -> Larry
mouse    -> mickey

```

This time we have got all the key/value pairs, including the multiple values associated with the key `Wall`.

To make life easier when dealing with duplicate keys, **DB_File** comes with a few utility methods.

The `get_dup()` Method

The `get_dup` method assists in reading duplicate values from BTREE databases. The method can take the following forms:

```

$count = $x->get_dup($key) ;
@list  = $x->get_dup($key) ;
%list  = $x->get_dup($key, 1) ;

```

In a scalar context the method returns the number of values associated with the key, `$key`.

In list context, it returns all the values which match `$key`. Note that the values will be returned in an apparently random order.

In list context, if the second parameter is present and evaluates TRUE, the method returns an associative array. The keys of the associative array correspond to the values that matched in the BTREE and the values of the array are a count of the number of times that particular value occurred in the BTREE.

So assuming the database created above, we can use `get_dup` like this:

```

use warnings ;
use strict ;
use DB_File ;

my ($filename, $x, %h) ;

$filename = "tree" ;

# Enable duplicate records
$DB_BTREE->{'flags'} = R_DUP ;

$x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
    or die "Cannot open $filename: $!\n";

my $cnt = $x->get_dup("Wall") ;
print "Wall occurred $cnt times\n" ;

my %hash = $x->get_dup("Wall", 1) ;
print "Larry is there\n" if $hash{'Larry'} ;

```

```

print "There are $hash{'Brick'} Brick Walls\n" ;

my @list = sort $x->get_dup("Wall") ;
print "Wall =>      [@list]\n" ;

@list = $x->get_dup("Smith") ;
print "Smith =>      [@list]\n" ;

@list = $x->get_dup("Dog") ;
print "Dog =>        [@list]\n" ;

```

and it will print:

```

Wall occurred 3 times
Larry is there
There are 2 Brick Walls
Wall =>      [Brick Brick Larry]
Smith =>      [John]
Dog =>        []

```

The `find_dup()` Method

```
$status = $X->find_dup($key, $value) ;
```

This method checks for the existence of a specific key/value pair. If the pair exists, the cursor is left pointing to the pair and the method returns 0. Otherwise the method returns a non-zero value.

Assuming the database from the previous example:

```

use warnings ;
use strict ;
use DB_File ;

my ($filename, $x, %h, $found) ;

$filename = "tree" ;

# Enable duplicate records
$DB_BTREE->{'flags'} = R_DUP ;

$x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
    or die "Cannot open $filename: $!\n";

$found = ( $x->find_dup("Wall", "Larry") == 0 ? "" : "not" ) ;
print "Larry Wall is $found there\n" ;

$found = ( $x->find_dup("Wall", "Harry") == 0 ? "" : "not" ) ;
print "Harry Wall is $found there\n" ;

undef $x ;
untie %h ;

```

prints this

```

Larry Wall is  there
Harry Wall is not there

```

The `del_dup()` Method

```
$status = $X->del_dup($key, $value) ;
```

This method deletes a specific key/value pair. It returns 0 if they exist and have been deleted successfully. Otherwise the method returns a non-zero value.

Again assuming the existence of the tree database

```
use warnings ;
```

```

use strict ;
use DB_File ;

my ($filename, $x, %h, $found) ;

$filename = "tree" ;

# Enable duplicate records
$DB_BTREE->{'flags'} = R_DUP ;

$x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
    or die "Cannot open $filename: $!\n";

$x->del_dup("Wall", "Larry") ;

$found = ( $x->find_dup("Wall", "Larry") == 0 ? "" : "not") ;
print "Larry Wall is $found there\n" ;

undef $x ;
untie %h ;

prints this

    Larry Wall is not there

```

Matching Partial Keys

The BTREE interface has a feature which allows partial keys to be matched. This functionality is *only* available when the seq method is used along with the R_CURSOR flag.

```
$x->seq($key, $value, R_CURSOR) ;
```

Here is the relevant quote from the dbopen man page where it defines the use of the R_CURSOR flag with seq:

Note, for the DB_BTREE access method, the returned key is not necessarily an exact match for the specified key. The returned key is the smallest key greater than or equal to the specified key, permitting partial key matches and range searches.

In the example script below, the match sub uses this feature to find and print the first matching key/value pair given a partial key.

```

use warnings ;
use strict ;
use DB_File ;
use Fcntl ;

my ($filename, $x, %h, $st, $key, $value) ;

sub match
{
    my $key = shift ;
    my $value = 0 ;
    my $orig_key = $key ;
    $x->seq($key, $value, R_CURSOR) ;
    print "$orig_key\t-> $key\t-> $value\n" ;
}

$filename = "tree" ;
unlink $filename ;

$x = tie %h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_BTREE
    or die "Cannot open $filename: $!\n";

```

```

# Add some key/value pairs to the file
$h{'mouse'} = 'mickey' ;
$h{'Wall'} = 'Larry' ;
$h{'Walls'} = 'Brick' ;
$h{'Smith'} = 'John' ;

$key = $value = 0 ;
print "IN ORDER\n" ;
for ($st = $x->seq($key, $value, R_FIRST) ;
    $st == 0 ;
    $st = $x->seq($key, $value, R_NEXT) )
    { print "$key    -> $value\n" }

print "\nPARTIAL MATCH\n" ;

match "Wa" ;
match "A" ;
match "a" ;

undef $x ;
untie %h ;

```

Here is the output:

```

IN ORDER
Smith -> John
Wall  -> Larry
Walls -> Brick
mouse -> mickey

PARTIAL MATCH
Wa -> Wall  -> Larry
A  -> Smith -> John
a  -> mouse -> mickey

```

DB_RECNO

DB_RECNO provides an interface to flat text files. Both variable and fixed length records are supported.

In order to make RECNO more compatible with Perl, the array offset for all RECNO arrays begins at 0 rather than 1 as in Berkeley DB.

As with normal Perl arrays, a RECNO array can be accessed using negative indexes. The index -1 refers to the last element of the array, -2 the second last, and so on. Attempting to access an element before the start of the array will raise a fatal run-time error.

The 'bval' Option

The operation of the bval option warrants some discussion. Here is the definition of bval from the Berkeley DB 1.85 recno manual page:

```

The delimiting byte to be used to mark the end of a
record for variable-length records, and the pad charac-
ter for fixed-length records.  If no value is speci-
fied, newlines ('\n') are used to mark the end of
variable-length records and fixed-length records are
padded with spaces.

```

The second sentence is wrong. In actual fact bval will only default to "\n" when the openinfo parameter in dbopen is NULL. If a non-NULL openinfo parameter is used at all, the value that happens to be in bval will be used. That means you always have to specify bval when making use of any of the options in the openinfo parameter. This documentation error will be fixed in the next release of Berkeley DB.

That clarifies the situation with regards Berkeley DB itself. What about **DB_File**? Well, the behavior defined in the quote above is quite useful, so **DB_File** conforms to it.

That means that you can specify other options (e.g. cachesize) and still have bval default to "\n" for variable length records, and space for fixed length records.

Also note that the bval option only allows you to specify a single byte as a delimiter.

A Simple Example

Here is a simple example that uses RECNO (if you are using a version of Perl earlier than 5.004_57 this example won't work — see [Extra RECNO Methods](#) for a workaround).

```
use warnings ;
use strict ;
use DB_File ;

my $filename = "text" ;
unlink $filename ;

my @h ;
tie @h, "DB_File", $filename, O_RDWR|O_CREAT, 0666, $DB_RECNO
    or die "Cannot open file 'text': $!\n" ;

# Add a few key/value pairs to the file
$h[0] = "orange" ;
$h[1] = "blue" ;
$h[2] = "yellow" ;

push @h, "green", "black" ;

my $elements = scalar @h ;
print "The array contains $elements entries\n" ;

my $last = pop @h ;
print "popped $last\n" ;

unshift @h, "white" ;
my $first = shift @h ;
print "shifted $first\n" ;

# Check for existence of a key
print "Element 1 Exists with value $h[1]\n" if $h[1] ;

# use a negative index
print "The last element is $h[-1]\n" ;
print "The 2nd last element is $h[-2]\n" ;

untie @h ;
```

Here is the output from the script:

```
The array contains 5 entries
popped black
shifted white
Element 1 Exists with value blue
The last element is green
The 2nd last element is yellow
```

Extra RECNO Methods

If you are using a version of Perl earlier than 5.004_57, the tied array interface is quite limited. In the example script above push, pop, shift, unshift or determining the array length will not work with a tied array.

To make the interface more useful for older versions of Perl, a number of methods are supplied with **DB_File** to simulate the missing array operations. All these methods are accessed via the object returned from the tie call.

Here are the methods:

`$X->push(list) ;`

Pushes the elements of `list` to the end of the array.

`$value = $X->pop ;`

Removes and returns the last element of the array.

`$X->shift`

Removes and returns the first element of the array.

`$X->unshift(list) ;`

Pushes the elements of `list` to the start of the array.

`$X->length`

Returns the number of elements in the array.

`$X->splice(offset, length, elements);`

Returns a splice of the the array.

Another Example

Here is a more complete example that makes use of some of the methods described above. It also makes use of the API interface directly (see [THE API INTERFACE](#)).

```
use warnings ;
use strict ;
my (@h, $H, $file, $i) ;
use DB_File ;
use Fcntl ;

$file = "text" ;

unlink $file ;

$H = tie @h, "DB_File", $file, O_RDWR|O_CREAT, 0666, $DB_RECNO
    or die "Cannot open file $file: $!\n" ;

# first create a text file to play with
$h[0] = "zero" ;
$h[1] = "one" ;
$h[2] = "two" ;
$h[3] = "three" ;
$h[4] = "four" ;

# Print the records in order.
#
# The length method is needed here because evaluating a tied
# array in a scalar context does not return the number of
# elements in the array.

print "\nORIGINAL\n" ;
foreach $i (0 .. $H->length - 1) {
    print "$i: $h[$i]\n" ;
}

# use the push & pop methods
$a = $H->pop ;
```

```

$H->push("last") ;
print "\nThe last record was [$a]\n" ;

# and the shift & unshift methods
$a = $H->shift ;
$H->unshift("first") ;
print "The first record was [$a]\n" ;

# Use the API to add a new record after record 2.
$i = 2 ;
$H->put($i, "Newbie", R_I AFTER) ;

# and a new record before record 1.
$i = 1 ;
$H->put($i, "New One", R_I BEFORE) ;

# delete record 3
$H->del(3) ;

# now print the records in reverse order
print "\nREVERSE\n" ;
for ($i = $H->length - 1 ; $i >= 0 ; -- $i)
    { print "$i: $h[$i]\n" }

# same again, but use the API functions instead
print "\nREVERSE again\n" ;
my ($s, $k, $v) = (0, 0, 0) ;
for ($s = $H->seq($k, $v, R_LAST) ;
     $s == 0 ;
     $s = $H->seq($k, $v, R_PREV))
    { print "$k: $v\n" }

undef $H ;
untie @h ;

```

and this is what it outputs:

```

ORIGINAL
0: zero
1: one
2: two
3: three
4: four

The last record was [four]
The first record was [zero]

REVERSE
5: last
4: three
3: Newbie
2: one
1: New One
0: first

REVERSE again
5: last
4: three
3: Newbie
2: one

```

```
1: New One
0: first
```

Notes:

1. Rather than iterating through the array, @h like this:

```
foreach $i (@h)
```

it is necessary to use either this:

```
foreach $i (0 .. $H->length - 1)
```

or this:

```
for ($a = $H->get($k, $v, R_FIRST) ;
    $a == 0 ;
    $a = $H->get($k, $v, R_NEXT) )
```

2. Notice that both times the put method was used the record index was specified using a variable, \$i, rather than the literal value itself. This is because put will return the record number of the inserted line via that parameter.

THE API INTERFACE

As well as accessing Berkeley DB using a tied hash or array, it is also possible to make direct use of most of the API functions defined in the Berkeley DB documentation.

To do this you need to store a copy of the object returned from the tie.

```
$db = tie %hash, "DB_File", "filename" ;
```

Once you have done that, you can access the Berkeley DB API functions as **DB_File** methods directly like this:

```
$db->put($key, $value, R_NOOVERWRITE) ;
```

Important: If you have saved a copy of the object returned from tie, the underlying database file will *not* be closed until both the tied variable is untied and all copies of the saved object are destroyed.

```
use DB_File ;
$db = tie %hash, "DB_File", "filename"
    or die "Cannot tie filename: $!" ;
...
undef $db ;
untie %hash ;
```

See [The untie\(\) Gotcha](#) for more details.

All the functions defined in [dbopen](#) are available except for close() and dbopen() itself. The **DB_File** method interface to the supported functions have been implemented to mirror the way Berkeley DB works whenever possible. In particular note that:

- The methods return a status value. All return 0 on success. All return -1 to signify an error and set \$! to the exact error code. The return code 1 generally (but not always) means that the key specified did not exist in the database.

Other return codes are defined. See below and in the Berkeley DB documentation for details. The Berkeley DB documentation should be used as the definitive source.
- Whenever a Berkeley DB function returns data via one of its parameters, the equivalent **DB_File** method does exactly the same.
- If you are careful, it is possible to mix API calls with the tied hash/array interface in the same piece of code. Although only a few of the methods used to implement the tied interface currently make use of the cursor, you should always assume that the cursor has been changed any time the tied

hash/array interface is used. As an example, this code will probably not do what you expect:

```
$X = tie %x, 'DB_File', $filename, O_RDWR|O_CREAT, 0777, $DB_BTREE
    or die "Cannot tie $filename: $!";

# Get the first key/value pair and set the cursor
$X->seq($key, $value, R_FIRST) ;

# this line will modify the cursor
$count = scalar keys %x ;

# Get the second key/value pair.
# oops, it didn't, it got the last key/value pair!
$X->seq($key, $value, R_NEXT) ;
```

The code above can be rearranged to get around the problem, like this:

```
$X = tie %x, 'DB_File', $filename, O_RDWR|O_CREAT, 0777, $DB_BTREE
    or die "Cannot tie $filename: $!";

# this line will modify the cursor
$count = scalar keys %x ;

# Get the first key/value pair and set the cursor
$X->seq($key, $value, R_FIRST) ;

# Get the second key/value pair.
# worked this time.
$X->seq($key, $value, R_NEXT) ;
```

All the constants defined in [dbopen](#) for use in the flags parameters in the methods defined below are also available. Refer to the Berkeley DB documentation for the precise meaning of the flags values.

Below is a list of the methods available.

`$status = $X->get($key, $value [, $flags]) ;`

Given a key (`$key`) this method reads the value associated with it from the database. The value read from the database is returned in the `$value` parameter.

If the key does not exist the method returns 1.

No flags are currently defined for this method.

`$status = $X->put($key, $value [, $flags]) ;`

Stores the key/value pair in the database.

If you use either the `R_IAFTER` or `R_IBEFORE` flags, the `$key` parameter will have the record number of the inserted key/value pair set.

Valid flags are `R_CURSOR`, `R_IAFTER`, `R_IBEFORE`, `R_NOOVERWRITE` and `R_SETCURSOR`.

`$status = $X->del($key [, $flags]) ;`

Removes all key/value pairs with key `$key` from the database.

A return code of 1 means that the requested key was not in the database.

`R_CURSOR` is the only valid flag at present.

`$status = $X->fd ;`

Returns the file descriptor for the underlying database.

See [Locking: The Trouble with fd](#) for an explanation for why you should not use `fd` to lock your database.

```
$status = $X->seq($key, $value, $flags) ;
```

This interface allows sequential retrieval from the database. See [dbopen](#) for full details.

Both the `$key` and `$value` parameters will be set to the key/value pair read from the database.

The flags parameter is mandatory. The valid flag values are `R_CURSOR`, `R_FIRST`, `R_LAST`, `R_NEXT` and `R_PREV`.

```
$status = $X->sync([ $flags ] ) ;
```

Flushes any cached buffers to disk.

`R_RECNO_SYNC` is the only valid flag at present.

DBM FILTERS

A DBM Filter is a piece of code that is be used when you *always* want to make the same transformation to all keys and/or values in a DBM database.

There are four methods associated with DBM Filters. All work identically, and each is used to install (or uninstall) a single DBM Filter. Each expects a single parameter, namely a reference to a sub. The only difference between them is the place that the filter is installed.

To summarise:

filter_store_key

If a filter has been installed with this method, it will be invoked every time you write a key to a DBM database.

filter_store_value

If a filter has been installed with this method, it will be invoked every time you write a value to a DBM database.

filter_fetch_key

If a filter has been installed with this method, it will be invoked every time you read a key from a DBM database.

filter_fetch_value

If a filter has been installed with this method, it will be invoked every time you read a value from a DBM database.

You can use any combination of the methods, from none, to all four.

All filter methods return the existing filter, if present, or `undef` in not.

To delete a filter pass `undef` to it.

The Filter

When each filter is called by Perl, a local copy of `$_` will contain the key or value to be filtered. Filtering is achieved by modifying the contents of `$_`. The return code from the filter is ignored.

An Example — the NULL termination problem.

Consider the following scenario. You have a DBM database that you need to share with a third-party C application. The C application assumes that *all* keys and values are NULL terminated. Unfortunately when Perl writes to DBM databases it doesn't use NULL termination, so your Perl application will have to manage NULL termination itself. When you write to the database you will have to use something like this:

```
$hash{"$key\0"} = "$value\0" ;
```

Similarly the NULL needs to be taken into account when you are considering the length of existing keys/values.

It would be much better if you could ignore the NULL terminations issue in the main application code and have a mechanism that automatically added the terminating NULL to all keys and values whenever you write

to the database and have them removed when you read from the database. As I'm sure you have already guessed, this is a problem that DBM Filters can fix very easily.

```
use warnings ;
use strict ;
use DB_File ;

my %hash ;
my $filename = "/tmp/filt" ;
unlink $filename ;

my $db = tie %hash, 'DB_File', $filename, O_CREAT|O_RDWR, 0666, $DB_HASH
    or die "Cannot open $filename: $!\n" ;

# Install DBM Filters
$db->filter_fetch_key  ( sub { s/\0$//      } ) ;
$db->filter_store_key   ( sub { $_ .= "\0"   } ) ;
$db->filter_fetch_value( sub { s/\0$//      } ) ;
$db->filter_store_value( sub { $_ .= "\0"   } ) ;

$hash{"abc"} = "def" ;
my $a = $hash{"ABC"} ;
# ...
undef $db ;
untie %hash ;
```

Hopefully the contents of each of the filters should be self-explanatory. Both "fetch" filters remove the terminating NULL, and both "store" filters add a terminating NULL.

Another Example — Key is a C int.

Here is another real-life example. By default, whenever Perl writes to a DBM database it always writes the key and value as strings. So when you use this:

```
$hash{12345} = "soemthing" ;
```

the key 12345 will get stored in the DBM database as the 5 byte string "12345". If you actually want the key to be stored in the DBM database as a C int, you will have to use `pack` when writing, and `unpack` when reading.

Here is a DBM Filter that does it:

```
use warnings ;
use strict ;
use DB_File ;
my %hash ;
my $filename = "/tmp/filt" ;
unlink $filename ;

my $db = tie %hash, 'DB_File', $filename, O_CREAT|O_RDWR, 0666, $DB_HASH
    or die "Cannot open $filename: $!\n" ;

$db->filter_fetch_key  ( sub { $_ = unpack("i", $_) } ) ;
$db->filter_store_key   ( sub { $_ = pack ("i", $_) } ) ;
$hash{123} = "def" ;
# ...
undef $db ;
untie %hash ;
```

This time only two filters have been used — we only need to manipulate the contents of the key, so it wasn't necessary to install any value filters.

HINTS AND TIPS

Locking: The Trouble with fd

Until version 1.72 of this module, the recommended technique for locking **DB_File** databases was to flock the filehandle returned from the "fd" function. Unfortunately this technique has been shown to be fundamentally flawed (Kudos to David Harris for tracking this down). Use it at your own peril!

The locking technique went like this.

```
$db = tie(%db, 'DB_File', '/tmp/foo.db', O_CREAT|O_RDWR, 0666)
    || die "dbcreat /tmp/foo.db $!";
$fd = $db->fd;
open(DB_FH, "+<&=$fd") || die "dup $!";
flock (DB_FH, LOCK_EX) || die "flock: $!";
...
$db{"Tom"} = "Jerry" ;
...
flock (DB_FH, LOCK_UN);
undef $db;
untie %db;
close (DB_FH);
```

In simple terms, this is what happens:

1. Use "tie" to open the database.
2. Lock the database with fd & flock.
3. Read & Write to the database.
4. Unlock and close the database.

Here is the crux of the problem. A side-effect of opening the **DB_File** database in step 2 is that an initial block from the database will get read from disk and cached in memory.

To see why this is a problem, consider what can happen when two processes, say "A" and "B", both want to update the same **DB_File** database using the locking steps outlined above. Assume process "A" has already opened the database and has a write lock, but it hasn't actually updated the database yet (it has finished step 2, but not started step 3 yet). Now process "B" tries to open the same database – step 1 will succeed, but it will block on step 2 until process "A" releases the lock. The important thing to notice here is that at this point in time both processes will have cached identical initial blocks from the database.

Now process "A" updates the database and happens to change some of the data held in the initial buffer. Process "A" terminates, flushing all cached data to disk and releasing the database lock. At this point the database on disk will correctly reflect the changes made by process "A".

With the lock released, process "B" can now continue. It also updates the database and unfortunately it too modifies the data that was in its initial buffer. Once that data gets flushed to disk it will overwrite some/all of the changes process "A" made to the database.

The result of this scenario is at best a database that doesn't contain what you expect. At worst the database will corrupt.

The above won't happen every time competing process update the same **DB_File** database, but it does illustrate why the technique should not be used.

Safe ways to lock a database

Starting with version 2.x, Berkeley DB has internal support for locking. The companion module to this one, **BerkeleyDB**, provides an interface to this locking functionality. If you are serious about locking Berkeley DB databases, I strongly recommend using **BerkeleyDB**.

If using **BerkeleyDB** isn't an option, there are a number of modules available on CPAN that can be used to implement locking. Each one implements locking differently and has different goals in mind. It is therefore worth knowing the difference, so that you can pick the right one for your application. Here are the three locking wrappers:

Tie::DB_Lock

A **DB_File** wrapper which creates copies of the database file for read access, so that you have a kind of a multiversioning concurrent read system. However, updates are still serial. Use for databases where reads may be lengthy and consistency problems may occur.

Tie::DB_LockFile

A **DB_File** wrapper that has the ability to lock and unlock the database while it is being used. Avoids the tie-before-flock problem by simply re-tie-ing the database when you get or drop a lock. Because of the flexibility in dropping and re-acquiring the lock in the middle of a session, this can be massaged into a system that will work with long updates and/or reads if the application follows the hints in the POD documentation.

DB_File::Lock

An extremely lightweight **DB_File** wrapper that simply flocks a lockfile before tie-ing the database and drops the lock after the untie. Allows one to use the same lockfile for multiple databases to avoid deadlock problems, if desired. Use for databases where updates are reads are quick and simple flock locking semantics are enough.

Sharing Databases With C Applications

There is no technical reason why a Berkeley DB database cannot be shared by both a Perl and a C application.

The vast majority of problems that are reported in this area boil down to the fact that C strings are NULL terminated, whilst Perl strings are not. See [DBM FILTERS](#) for a generic way to work around this problem.

Here is a real example. Netscape 2.0 keeps a record of the locations you visit along with the time you last visited them in a DB_HASH database. This is usually stored in the file `~/.netscape/history.db`. The key field in the database is the location string and the value field is the time the location was last visited stored as a 4 byte binary value.

If you haven't already guessed, the location string is stored with a terminating NULL. This means you need to be careful when accessing the database.

Here is a snippet of code that is loosely based on Tom Christiansen's *ggh* script (available from your nearest CPAN archive in [authors/id/TOMC/scripts/nshist.gz](#)).

```
use warnings ;
use strict ;
use DB_File ;
use Fcntl ;

my ($dotdir, $HISTORY, %hist_db, $href, $binary_time, $date) ;
$dotdir = $ENV{HOME} || $ENV{LOGNAME};
$HISTORY = "$dotdir/.netscape/history.db";

tie %hist_db, 'DB_File', $HISTORY
    or die "Cannot open $HISTORY: $!\n" ;;

# Dump the complete database
while ( ($href, $binary_time) = each %hist_db ) {
    # remove the terminating NULL
    $href =~ s/\x00$// ;

    # convert the binary time into a user friendly string
```

```

        $date = localtime unpack("V", $binary_time);
        print "$date $href\n" ;
    }

    # check for the existence of a specific key
    # remember to add the NULL
    if ( $binary_time = $hist_db{"http://mox.perl.com/\x00"} ) {
        $date = localtime unpack("V", $binary_time) ;
        print "Last visited mox.perl.com on $date\n" ;
    }
    else {
        print "Never visited mox.perl.com\n"
    }

    untie %hist_db ;

```

The `untie()` Gotcha

If you make use of the Berkeley DB API, it is *very* strongly recommended that you read [The `untie` Gotcha](#).

Even if you don't currently make use of the API interface, it is still worth reading it.

Here is an example which illustrates the problem from a **DB_File** perspective:

```

use DB_File ;
use Fcntl ;

my %x ;
my $X ;

$X = tie %x, 'DB_File', 'tst.fil' , O_RDWR|O_TRUNC
    or die "Cannot tie first time: $!" ;

$x{123} = 456 ;

untie %x ;

tie %x, 'DB_File', 'tst.fil' , O_RDWR|O_CREAT
    or die "Cannot tie second time: $!" ;

untie %x ;

```

When run, the script will produce this error message:

```
Cannot tie second time: Invalid argument at bad.file line 14.
```

Although the error message above refers to the second `tie()` statement in the script, the source of the problem is really with the `untie()` statement that precedes it.

Having read [perl tie](#) you will probably have already guessed that the error is caused by the extra copy of the tied object stored in `$X`. If you haven't, then the problem boils down to the fact that the **DB_File** destructor, `DESTROY`, will not be called until *all* references to the tied object are destroyed. Both the tied variable, `%x`, and `$X` above hold a reference to the object. The call to `untie()` will destroy the first, but `$X` still holds a valid reference, so the destructor will not get called and the database file *tst.fil* will remain open. The fact that Berkeley DB then reports the attempt to open a database that is already open via the catch-all "Invalid argument" doesn't help.

If you run the script with the `-w` flag the error message becomes:

```
untie attempted while 1 inner references still exist at bad.file line 12.
Cannot tie second time: Invalid argument at bad.file line 14.
```

which pinpoints the real problem. Finally the script can now be modified to fix the original problem by destroying the API object before the `untie`:

```
...
$x{123} = 456 ;
undef $X ;
untie %x ;

$X = tie %x, 'DB_File', 'tst.fil' , O_RDWR|O_CREAT
...
```

COMMON QUESTIONS

Why is there Perl source in my database?

If you look at the contents of a database file created by DB_File, there can sometimes be part of a Perl script included in it.

This happens because Berkeley DB uses dynamic memory to allocate buffers which will subsequently be written to the database file. Being dynamic, the memory could have been used for anything before DB malloced it. As Berkeley DB doesn't clear the memory once it has been allocated, the unused portions will contain random junk. In the case where a Perl script gets written to the database, the random junk will correspond to an area of dynamic memory that happened to be used during the compilation of the script.

Unless you don't like the possibility of there being part of your Perl scripts embedded in a database file, this is nothing to worry about.

How do I store complex data structures with DB_File?

Although DB_File cannot do this directly, there is a module which can layer transparently over DB_File to accomplish this feat.

Check out the MLDBM module, available on CPAN in the directory *modules/by-module/MLDBM*.

What does "Invalid Argument" mean?

You will get this error message when one of the parameters in the tie call is wrong. Unfortunately there are quite a few parameters to get wrong, so it can be difficult to figure out which one it is.

Here are a couple of possibilities:

1. Attempting to reopen a database without closing it.
2. Using the O_WRONLY flag.

What does "Bareword 'DB_File' not allowed" mean?

You will encounter this particular error message when you have the strict 'subs' pragma (or the full strict pragma) in your script. Consider this script:

```
use warnings ;
use strict ;
use DB_File ;
my %x ;
tie %x, DB_File, "filename" ;
```

Running it produces the error in question:

```
Bareword "DB_File" not allowed while "strict subs" in use
```

To get around the error, place the word DB_File in either single or double quotes, like this:

```
tie %x, "DB_File", "filename" ;
```

Although it might seem like a real pain, it is really worth the effort of having a use strict in all your scripts.

REFERENCES

Articles that are either about **DB_File** or make use of it.

1. *Full-Text Searching in Perl*, Tim Kientzle (tkientzle@ddj.com), Dr. Dobbs's Journal, Issue 295, January 1999, pp 34–41

HISTORY

Moved to the Changes file.

BUGS

Some older versions of Berkeley DB had problems with fixed length records using the RECNO file format. This problem has been fixed since version 1.85 of Berkeley DB.

I am sure there are bugs in the code. If you do find any, or can suggest any enhancements, I would welcome your comments.

AVAILABILITY

DB_File comes with the standard Perl source distribution. Look in the directory *ext/DB_File*. Given the amount of time between releases of Perl the version that ships with Perl is quite likely to be out of date, so the most recent version can always be found on CPAN (see [CPAN](#) for details), in the directory *modules/by-module/DB_File*.

This version of **DB_File** will work with either version 1.x, 2.x or 3.x of Berkeley DB, but is limited to the functionality provided by version 1.

The official web site for Berkeley DB is <http://www.sleepycat.com>. All versions of Berkeley DB are available there.

Alternatively, Berkeley DB version 1 is available at your nearest CPAN archive in *src/misc/db.1.85.tar.gz*.

If you are running IRIX, then get Berkeley DB version 1 from <http://reality.sgi.com/ariel>. It has the patches necessary to compile properly on IRIX 5.3.

COPYRIGHT

Copyright (c) 1995–2002 Paul Marquess. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Although **DB_File** is covered by the Perl license, the library it makes use of, namely Berkeley DB, is not. Berkeley DB has its own copyright and its own license. Please take the time to read it.

Here are a few words taken from the Berkeley DB FAQ (at <http://www.sleepycat.com>) regarding the license:

Do I have to license DB to use it in Perl scripts?

No. The Berkeley DB license requires that software that uses Berkeley DB be freely redistributable. In the case of Perl, that software is Perl, and not your scripts. Any Perl scripts that you write are your property, including scripts that make use of Berkeley DB. Neither the Perl license nor the Berkeley DB license place any restriction on what you may do with them.

If you are in any doubt about the license situation, contact either the Berkeley DB authors or the author of **DB_File**. See *"AUTHOR"* for details.

SEE ALSO

[perl\(1\)](#), [dbopen\(3\)](#), [hash\(3\)](#), [recno\(3\)](#), [btree\(3\)](#), [dbmfilter](#)

AUTHOR

The DB_File interface was written by Paul Marquess <Paul.Marquess@btinternet.com>. Questions about the DB system itself may be addressed to <db@sleepycat.com>.

NAME

Devel::DProf – a Perl code profiler

SYNOPSIS

```
perl5 -d:DProf test.pl
```

DESCRIPTION

The Devel::DProf package is a Perl code profiler. This will collect information on the execution time of a Perl script and of the subs in that script. This information can be used to determine which subroutines are using the most time and which subroutines are being called most often. This information can also be used to create an execution graph of the script, showing subroutine relationships.

To profile a Perl script run the perl interpreter with the **-d** debugging switch. The profiler uses the debugging hooks. So to profile script *test.pl* the following command should be used:

```
perl5 -d:DProf test.pl
```

When the script terminates (or when the output buffer is filled) the profiler will dump the profile information to a file called *mon.out*. A tool like *dprofpp* can be used to interpret the information which is in that profile.

The following command will print the top 15 subroutines which used the most time:

```
dprofpp
```

To print an execution graph of the subroutines in the script use the following command:

```
dprofpp -T
```

Consult *dprofpp* for other options.

PROFILE FORMAT

The old profile is a text file which looks like this:

```
#fOrTyTwo
$hz=100;
$XS_VERSION='DProf 19970606';
# All values are given in HZ
$rrun_utime=2; $rrun_stime=0; $rrun_rtime=7
PART2
+ 26 28 566822884 DynaLoader::import
- 26 28 566822884 DynaLoader::import
+ 27 28 566822885 main::bar
- 27 28 566822886 main::bar
+ 27 28 566822886 main::baz
+ 27 28 566822887 main::bar
- 27 28 566822888 main::bar
[....]
```

The first line is the magic number. The second line is the hertz value, or clock ticks, of the machine where the profile was collected. The third line is the name and version identifier of the tool which created the profile. The fourth line is a comment. The fifth line contains three variables holding the user time, system time, and realtime of the process while it was being profiled. The sixth line indicates the beginning of the sub entry/exit profile section.

The columns in **PART2** are:

```
sub entry(+)/exit(-) mark
app's user time at sub entry/exit mark, in ticks
app's system time at sub entry/exit mark, in ticks
app's realtime at sub entry/exit mark, in ticks
fully-qualified sub name, when possible
```

With newer perls another format is used, which may look like this:

```
#fOrTyTwoO
$hz=10000;
$XS_VERSION='DProf 19971213';
# All values are given in HZ
$over_untime=5917; $over_stime=0; $over_rtime=5917;
$over_tests=10000;
$rrun_untime=1284; $rrun_stime=0; $rrun_rtime=1284;
$total_marks=6;

PART2
@ 406 0 406
& 2 main bar
+ 2
@ 456 0 456
- 2
@ 1 0 1
& 3 main baz
+ 3
@ 141 0 141
+ 2
@ 141 0 141
- 2
@ 1 0 1
& 4 main foo
+ 4
@ 142 0 142
+ & Devel::DProf::write
@ 5 0 5
- & Devel::DProf::write
```

(with high value of `$ENV{PERL_DPROF_TICKS}`).

New `$over_*` values show the measured overhead of making `$over_tests` calls to the profiler. These values are used by the profiler to subtract the overhead from the runtimes.

The lines starting with `@` mark time passed from the previous `@` line. The lines starting with `&` introduce new subroutine *id* and show the package and the subroutine name of this *id*. Lines starting with `+`, `-` and `*` mark entering and exit of subroutines by *ids*, and `goto &subr`.

The *old-style* `+-` and `--` lines are used to mark the overhead related to writing to profiler-output file.

AUTOLOAD

When `Devel::DProf` finds a call to an `&AUTOLOAD` subroutine it looks at the `$AUTOLOAD` variable to find the real name of the sub being called. See [Autoloading in perlsub](#).

ENVIRONMENT

`PERL_DPROF_BUFFER` sets size of output buffer in words. Defaults to `2*14`.

`PERL_DPROF_TICKS` sets number of ticks per second on some systems where a replacement for `times()` is used. Defaults to the value of `HZ` macro.

`PERL_DPROF_OUT_FILE_NAME` sets the name of the output file. If not set, defaults to `tmon.out`.

BUGS

Builtin functions cannot be measured by `Devel::DProf`.

With a newer Perl `DProf` relies on the fact that the numeric slot of `$DB::sub` contains an address of a subroutine. Excessive manipulation of this variable may overwrite this slot, as in

```
$DB::sub = 'current_sub';  
...  
$addr = $DB::sub + 0;
```

will set this numeric slot to numeric value of the string `current_sub`, i.e., to `.` This will cause a segfault on the exit from this subroutine. Note that the first assignment above does not change the numeric slot (it will *mark* it as invalid, but will not write over it).

Mail bug reports and feature requests to the perl5-porters mailing list at [<perl5-porters@perl.org>](mailto:perl5-porters@perl.org).

SEE ALSO

[perl](#), [dprofpp](#), `times(2)`

NAME

Devel::Peek – A data debugging tool for the XS programmer

SYNOPSIS

```
use Devel::Peek;
Dump( $a );
Dump( $a, 5 );
DumpArray( 5, $a, $b, ... );
mstat "Point 5";

use Devel::Peek ':opd=st';
```

DESCRIPTION

Devel::Peek contains functions which allows raw Perl datatypes to be manipulated from a Perl script. This is used by those who do XS programming to check that the data they are sending from C to Perl looks as they think it should look. The trick, then, is to know what the raw datatype is supposed to look like when it gets to Perl. This document offers some tips and hints to describe good and bad raw data.

It is very possible that this document will fall far short of being useful to the casual reader. The reader is expected to understand the material in the first few sections of [perlguts](#).

Devel::Peek supplies a `Dump()` function which can dump a raw Perl datatype, and `mstat("marker")` function to report on memory usage (if perl is compiled with corresponding option). The function `DeadCode()` provides statistics on the data "frozen" into inactive CV. Devel::Peek also supplies `SvREFCNT()`, `SvREFCNT_inc()`, and `SvREFCNT_dec()` which can query, increment, and decrement reference counts on SVs. This document will take a passive, and safe, approach to data debugging and for that it will describe only the `Dump()` function.

Function `DumpArray()` allows dumping of multiple values (useful when you need to analyze returns of functions).

The global variable `$Devel::Peek::pv_limit` can be set to limit the number of character printed in various string values. Setting it to 0 means no limit.

If use `Devel::Peek` directive has a `:opd=FLAGS` argument, this switches on debugging of opcode dispatch. `FLAGS` should be a combination of `s`, `t`, and `P` (see `-D` flags in [perlrun](#)). `:opd` is a shortcut for `:opd=st`.

Runtime debugging

`CvGV($cv)` return one of the globs associated to a subroutine reference `$cv`.

`debug_flags()` returns a string representation of `$_D` (similar to what is allowed for `-D` flag). When called with a numeric argument, sets `$_D` to the corresponding value. When called with an argument of the form `"flags-flags"`, set on/off bits of `$_D` corresponding to letters before/after `-`. (The returned value is for `$_D` before the modification.)

`runops_debug()` returns true if the current *opcode dispatcher* is the debugging one. When called with an argument, switches to debugging or non-debugging dispatcher depending on the argument (active for newly-entered subs/etc only). (The returned value is for the dispatcher before the modification.)

Memory footprint debugging

When perl is compiled with support for memory footprint debugging (default with Perl's `malloc()`), Devel::Peek provides an access to this API.

Use `mstat()` function to emit a memory state statistic to the terminal. For more information on the format of output of `mstat()` see [Using \\$ENV{PERL_DEBUG_MSTATS}](#).

Three additional functions allow access to this statistic from Perl. First, use `mstats_fillhash(%hash)` to get the information contained in the output of `mstat()` into `%hash`. The field of this hash are

```
minbucket nbuckets sbrk_good sbrk_slack sbrked_remains sbrks start_slack
topbucket topbucket_ev topbucket_odd total total_chain total_sbrk totfree
```

Two additional fields `free`, `used` contain array references which provide per-bucket count of free and used chunks. Two other fields `mem_size`, `available_size` contain array references which provide the information about the allocated size and usable size of chunks in each bucket. Again, see [Using \\$ENV{PERL_DEBUG_MSTATS}](#) for details.

Keep in mind that only the first several "odd-numbered" buckets are used, so the information on size of the "odd-numbered" buckets which are not used is probably meaningless.

The information in

```
mem_size available_size minbucket nbuckets
```

is the property of a particular build of perl, and does not depend on the current process. If you do not provide the optional argument to the functions `mstats_fillhash()`, `fill_mstats()`, `mstats2hash()`, then the information in fields `mem_size`, `available_size` is not updated.

`fill_mstats($buf)` is a much cheaper call (both speedwise and memory-wise) which collects the statistic into `$buf` in machine-readable form. At a later moment you may need to call `mstats2hash($buf, %hash)` to use this information to fill `%hash`.

All three APIs `fill_mstats($buf)`, `mstats_fillhash(%hash)`, and `mstats2hash($buf, %hash)` are designed to allocate no memory if used *the second time* on the same `$buf` and/or `%hash`.

So, if you want to collect memory info in a cycle, you may call

```
$#buf = 999;
fill_mstats($_) for @buf;
mstats_fillhash(%report, 1);           # Static info too

foreach (@buf) {
    # Do something...
    fill_mstats $_;                   # Collect statistic
}
foreach (@buf) {
    mstats2hash($_, %report);         # Preserve static info
    # Do something with %report
}
```

EXAMPLES

The following examples don't attempt to show everything as that would be a monumental task, and, frankly, we don't want this manpage to be an internals document for Perl. The examples do demonstrate some basics of the raw Perl datatypes, and should suffice to get most determined people on their way. There are no guidewires or safety nets, nor blazed trails, so be prepared to travel alone from this point and on and, if at all possible, don't fall into the quicksand (it's bad for business).

Oh, one final bit of advice: take [perlguys](#) with you. When you return we expect to see it well-thumbed.

A simple scalar string

Let's begin by looking a simple scalar which is holding a string.

```
use Devel::Peek;
$a = "hello";
Dump $a;
```

The output:

```
SV = PVIV(0xbc288)
  REFCNT = 1
  FLAGS = (POK,pPOK)
```

```
IV = 0
PV = 0xb2048 "hello"\0
CUR = 5
LEN = 6
```

This says \$a is an SV, a scalar. The scalar is a PVIV, a string. Its reference count is 1. It has the POK flag set, meaning its current PV field is valid. Because POK is set we look at the PV item to see what is in the scalar. The \0 at the end indicate that this PV is properly NUL-terminated. If the FLAGS had been IOK we would look at the IV item. CUR indicates the number of characters in the PV. LEN indicates the number of bytes requested for the PV (one more than CUR, in this case, because LEN includes an extra byte for the end-of-string marker).

A simple scalar number

If the scalar contains a number the raw SV will be leaner.

```
use Devel::Peek;
$a = 42;
Dump $a;
```

The output:

```
SV = IV(0xbc818)
REFCNT = 1
FLAGS = (IOK,pIOK)
IV = 42
```

This says \$a is an SV, a scalar. The scalar is an IV, a number. Its reference count is 1. It has the IOK flag set, meaning it is currently being evaluated as a number. Because IOK is set we look at the IV item to see what is in the scalar.

A simple scalar with an extra reference

If the scalar from the previous example had an extra reference:

```
use Devel::Peek;
$a = 42;
$b = \ $a;
Dump $a;
```

The output:

```
SV = IV(0xbe860)
REFCNT = 2
FLAGS = (IOK,pIOK)
IV = 42
```

Notice that this example differs from the previous example only in its reference count. Compare this to the next example, where we dump \$b instead of \$a.

A reference to a simple scalar

This shows what a reference looks like when it references a simple scalar.

```
use Devel::Peek;
$a = 42;
$b = \ $a;
Dump $b;
```

The output:

```
SV = RV(0xf041c)
REFCNT = 1
FLAGS = (ROK)
```

```

RV = 0xbab08
SV = IV(0xbe860)
REFCNT = 2
FLAGS = (IOK,pIOK)
IV = 42

```

Starting from the top, this says \$b is an SV. The scalar is an RV, a reference. It has the ROK flag set, meaning it is a reference. Because ROK is set we have an RV item rather than an IV or PV. Notice that Dump follows the reference and shows us what \$b was referencing. We see the same \$a that we found in the previous example.

Note that the value of RV coincides with the numbers we see when we stringify \$b. The addresses inside RV() and IV() are addresses of X*** structure which holds the current state of an SV. This address may change during lifetime of an SV.

A reference to an array

This shows what a reference to an array looks like.

```

use Devel::Peek;
$a = [42];
Dump $a;

```

The output:

```

SV = RV(0xf041c)
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0xb2850
SV = PVAV(0xbd448)
  REFCNT = 1
  FLAGS = ( )
  IV = 0
  NV = 0
  ARRAY = 0xb2048
  ALLOC = 0xb2048
  FILL = 0
  MAX = 0
  ARYLEN = 0x0
  FLAGS = (REAL)
Elt No. 0 0xb5658
SV = IV(0xbe860)
  REFCNT = 1
  FLAGS = (IOK,pIOK)
  IV = 42

```

This says \$a is an SV and that it is an RV. That RV points to another SV which is a PVAV, an array. The array has one element, element zero, which is another SV. The field FILL above indicates the last element in the array, similar to \$#\$a.

If \$a pointed to an array of two elements then we would see the following.

```

use Devel::Peek 'Dump';
$a = [42,24];
Dump $a;

```

The output:

```

SV = RV(0xf041c)
  REFCNT = 1
  FLAGS = (ROK)

```



```

RV = 0xb2850
SV = PVAV(0xbd448)
REFCNT = 1
FLAGS = ()
IV = 0
NV = 0
ARRAY = 0xb2048
ALLOC = 0xb2048
FILL = 0
MAX = 0
ARYLEN = 0x0
FLAGS = (REAL)
Elt No. 0 0xb5658
SV = IV(0xbe860)
REFCNT = 1
FLAGS = (IOK,pIOK)
IV = 42
Elt No. 1 0xb5680
SV = IV(0xbe818)
REFCNT = 1
FLAGS = (IOK,pIOK)
IV = 24

```

Note that Dump will not report *all* the elements in the array, only several first (depending on how deep it already went into the report tree).

A reference to a hash

The following shows the raw form of a reference to a hash.

```

use Devel::Peek;
$a = {hello=>42};
Dump $a;

```

The output:

```

SV = RV(0xf041c)
REFCNT = 1
FLAGS = (ROK)
RV = 0xb2850
SV = PVHV(0xbd448)
REFCNT = 1
FLAGS = ()
NV = 0
ARRAY = 0xbd748
KEYS = 1
FILL = 1
MAX = 7
RITER = -1
EITER = 0x0
Elt "hello" => 0xbaaf0
SV = IV(0xbe860)
REFCNT = 1
FLAGS = (IOK,pIOK)
IV = 42

```

This shows `$a` is a reference pointing to an SV. That SV is a PVHV, a hash. Fields RITER and EITER are used by *each*.

Dumping a large array or hash

The `Dump()` function, by default, dumps up to 4 elements from a toplevel array or hash. This number can be increased by supplying a second argument to the function.

```
use Devel::Peek;
$a = [10,11,12,13,14];
Dump $a;
```

Notice that `Dump()` prints only elements 10 through 13 in the above code. The following code will print all of the elements.

```
use Devel::Peek 'Dump';
$a = [10,11,12,13,14];
Dump $a, 5;
```

A reference to an SV which holds a C pointer

This is what you really need to know as an XS programmer, of course. When an XSUB returns a pointer to a C structure that pointer is stored in an SV and a reference to that SV is placed on the XSUB stack. So the output from an XSUB which uses something like the `T_PTROBJ` map might look something like this:

```
SV = RV(0xf381c)
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0xb8ad8
SV = PVMG(0xbb3c8)
  REFCNT = 1
  FLAGS = (OBJECT, IOK, pIOK)
  IV = 729160
  NV = 0
  PV = 0
  STASH = 0xc1d10      "CookBookB::Opaque"
```

This shows that we have an SV which is an RV. That RV points at another SV. In this case that second SV is a PVMG, a blessed scalar. Because it is blessed it has the `OBJECT` flag set. Note that an SV which holds a C pointer also has the `IOK` flag set. The `STASH` is set to the package name which this SV was blessed into.

The output from an XSUB which uses something like the `T_PTRREF` map, which doesn't bless the object, might look something like this:

```
SV = RV(0xf381c)
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0xb8ad8
SV = PVMG(0xbb3c8)
  REFCNT = 1
  FLAGS = (IOK, pIOK)
  IV = 729160
  NV = 0
  PV = 0
```

A reference to a subroutine

Looks like this:

```
SV = RV(0x798ec)
  REFCNT = 1
  FLAGS = (TEMP, ROK)
  RV = 0x1d453c
```

```

SV = PVCV(0x1c768c)
REFCNT = 2
FLAGS = ()
IV = 0
NV = 0
COMP_STASH = 0x31068 "main"
START = 0xb20e0
ROOT = 0xbece0
XSUB = 0x0
XSUBANY = 0
GVGV::GV = 0x1d44e8 "MY" :: "top_targets"
FILE = "(eval 5)"
DEPTH = 0
PADLIST = 0x1c9338

```

This shows that

- the subroutine is not an XSUB (since START and ROOT are non-zero, and XSUB is zero);
- that it was compiled in the package main;
- under the name MY::top_targets;
- inside a 5th eval in the program;
- it is not currently executed (see DEPTH);
- it has no prototype (PROTOTYPE field is missing).

EXPORTS

Dump, mstat, DeadCode, DumpArray, DumpWithOP and DumpProg, fill_mstats, mstats_fillhash, mstats2hash by default. Additionally available SvREFCNT, SvREFCNT_inc and SvREFCNT_dec.

BUGS

Readers have been known to skip important parts of [perlguts](#), causing much frustration for all.

AUTHOR

Ilya Zakharevich ilya@math.ohio-state.edu

Copyright (c) 1995–98 Ilya Zakharevich. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Author of this software makes no claim whatsoever about suitability, reliability, edability, editability or usability of this product, and should not be kept liable for any damage resulting from the use of it. If you can use it, you are in luck, if not, I should not be kept responsible. Keep a handy copy of your backup tape at hand.

SEE ALSO

[perlguts](#), and [perlguts](#), again.

NAME

Perl/Pollution/Portability

SYNOPSIS

```
Devel::PPPort::WriteFile() ; # defaults to ./ppport.h
Devel::PPPort::WriteFile('someheader.h') ;
```

DESCRIPTION

Perl has changed over time, gaining new features, new functions, increasing its flexibility, and reducing the impact on the C namespace environment (reduced pollution). The header file, typically `ppport.h`, written by this module attempts to bring some of the newer Perl features to older versions of Perl, so that you can worry less about keeping track of old releases, but users can still reap the benefit.

Why you should use `ppport.h` in modern code: so that your code will work with the widest range of Perl interpreters possible, without significant additional work.

Why you should attempt older code to fully use `ppport.h`: because the reduced pollution of newer Perl versions is an important thing, so important that the old polluting ways of original Perl modules will not be supported very far into the future, and your module will almost certainly break! By adapting to it now, you'll gained compatibility and a sense of having done the electronic ecology some good.

How to use `ppport.h`: Don't direct the user to download `Devel::PPPort`, and don't make `ppport.h` optional. Rather, just take the most recent copy of `ppport.h` that you can find (probably in `Devel::PPPort` on CPAN), copy it into your project, adjust your project to use it, and distribute the header along with your module.

`Devel::PPPort` contains a single function, called `WriteFile`. It's purpose is to write a 'C' header file that is used when writing XS modules. The file contains a series of macros that allow XS modules to be built using older versions of Perl.

This module is used by `h2xs` to write the file *ppport.h*.

WriteFile

`WriteFile` takes a zero or one parameters. When called with one parameter it expects to be passed a filename. When called with no parameters, it defaults to the filename `./ppport.h`.

The function returns `TRUE` if the file was written successfully. Otherwise it returns `FALSE`.

ppport.h

The file written by this module, typically `ppport.h`, provides access to the following Perl API if not already available (and in some cases [*] even if available, access to a fixed interface):

```
aMY_CXT
aMY_CXT_
_aMY_CXT
aTHX
aTHX_
AvFILLp
boolSV(b)
DEFSV
dMY_CXT
dMY_CXT_SV
dNOOP
dTHR
dTHX
dTHXa
dTHXoa
ERRSV
gv_stashpvn(str,len,flags)
```

INT2PTR(type,int)
IVdf
MY_CXT
MY_CXT_INIT
newCONSTSUB(stash,name,sv)
newRV_inc(sv)
newRV_noinc(sv)
newSVpvn(data,len)
NOOP
NV
NVef
NVff
NVgf
PERL_REVISION
PERL_SUBVERSION
PERL_UNUSED_DECL
PERL_UNUSED_DECL
PERL_VERSION
PL_compiling
PL_copline
PL_curcop
PL_curstash
PL_defgv
PL_dirty
PL_hints
PL_na
PL_perldb
PL_rsfilters
PL_rsfpv
PL_stdin
PL_Sv
PL_sv_no
PL_sv_undef
PL_sv_yes
pMY_CXT
pMY_CXT_
_pMY_CXT
pTHX
pTHX_
PTR2IV(ptr)
PTR2NV(ptr)
PTR2ul(ptr)
PTR2UV(ptr)
SAVE_DEFSV
START_MY_CXT
SvPVbyte(sv,lp) [*]
UVof
UVSIZE
UVuf
UVxf
UVXf

AUTHOR

Version 1.x of Devel::PPPort was written by Kenneth Albanowski.

Version 2.x was ported to the Perl core by Paul Marquess.

SEE ALSO

See [h2xs](#).

NAME

Devel::SelfStubber – generate stubs for a SelfLoading module

SYNOPSIS

To generate just the stubs:

```
use Devel::SelfStubber;  
Devel::SelfStubber->stub('MODULENAME', 'MY_LIB_DIR');
```

or to generate the whole module with stubs inserted correctly

```
use Devel::SelfStubber;  
$Devel::SelfStubber::JUST_STUBS=0;  
Devel::SelfStubber->stub('MODULENAME', 'MY_LIB_DIR');
```

MODULENAME is the Perl module name, e.g. Devel::SelfStubber, NOT 'Devel/SelfStubber' or 'Devel/SelfStubber.pm'.

MY_LIB_DIR defaults to '.' if not present.

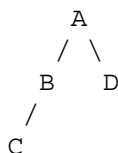
DESCRIPTION

Devel::SelfStubber prints the stubs you need to put in the module before the `__DATA__` token (or you can get it to print the entire module with stubs correctly placed). The stubs ensure that if a method is called, it will get loaded. They are needed specifically for inherited autoloading methods.

This is best explained using the following example:

Assume four classes, A,B,C & D.

A is the root class, B is a subclass of A, C is a subclass of B, and D is another subclass of A.



If D calls an autoloading method 'foo' which is defined in class A, then the method is loaded into class A, then executed. If C then calls method 'foo', and that method was reimplemented in class B, but set to be autoloading, then the lookup mechanism never gets to the AUTOLOAD mechanism in B because it first finds the method already loaded in A, and so erroneously uses that. If the method foo had been stubbed in B, then the lookup mechanism would have found the stub, and correctly loaded and used the sub from B.

So, for classes and subclasses to have inheritance correctly work with autoloading, you need to ensure stubs are loaded.

The SelfLoader can load stubs automatically at module initialization with the statement

`'SelfLoader->load_stubs()'`; but you may wish to avoid having the stub loading overhead associated with your initialization (though note that the `SelfLoader::load_stubs` method will be called sooner or later – at latest when the first sub is being autoloading). In this case, you can put the sub stubs before the `__DATA__` token. This can be done manually, but this module allows automatic generation of the stubs.

By default it just prints the stubs, but you can set the global `$Devel::SelfStubber::JUST_STUBS` to 0 and it will print out the entire module with the stubs positioned correctly.

At the very least, this is useful to see what the SelfLoader thinks are stubs – in order to ensure future versions of the SelfStubber remain in step with the SelfLoader, the SelfStubber actually uses the SelfLoader to determine which stubs are needed.

NAME

`diagnostics` – Perl compiler pragma to force verbose warning diagnostics

`splain` – standalone program to do the same thing

SYNOPSIS

As a pragma:

```
use diagnostics;
use diagnostics -verbose;

enable diagnostics;
disable diagnostics;
```

As a program:

```
perl program 2>diag.out
splain [-v] [-p] diag.out
```

DESCRIPTION

The `diagnostics` Pragma

This module extends the terse diagnostics normally emitted by both the perl compiler and the perl interpreter, augmenting them with the more explicative and endearing descriptions found in [perl_{ldiag}](#). Like the other pragmata, it affects the compilation phase of your program rather than merely the execution phase.

To use in your program as a pragma, merely invoke

```
use diagnostics;
```

at the start (or near the start) of your program. (Note that this *does* enable perl's `-w` flag.) Your whole compilation will then be subject(ed :-)) to the enhanced diagnostics. These still go out **STDERR**.

Due to the interaction between runtime and compiletime issues, and because it's probably not a very good idea anyway, you may not use `no diagnostics` to turn them off at compiletime. However, you may control their behaviour at runtime using the `disable()` and `enable()` methods to turn them off and on respectively.

The `-verbose` flag first prints out the [perl_{ldiag}](#) introduction before any other diagnostics. The `$diagnostics::PRETTY` variable can generate nicer escape sequences for pagers.

Warnings dispatched from perl itself (or more accurately, those that match descriptions found in [perl_{ldiag}](#)) are only displayed once (no duplicate descriptions). User code generated warnings ala `warn()` are unaffected, allowing duplicate user messages to be displayed.

The `splain` Program

While apparently a whole nuther program, *splain* is actually nothing more than a link to the (executable) *diagnostics.pm* module, as well as a link to the *diagnostics.pod* documentation. The `-v` flag is like the `use diagnostics -verbose` directive. The `-p` flag is like the `$diagnostics::PRETTY` variable. Since you're post-processing with *splain*, there's no sense in being able to `enable()` or `disable()` processing.

Output from *splain* is directed to **STDOUT**, unlike the pragma.

EXAMPLES

The following file is certain to trigger a few errors at both runtime and compiletime:

```
use diagnostics;
print NOWHERE "nothing\n";
print STDERR "\n\tThis message should be unadorned.\n";
warn "\tThis is a user warning";
print "\nDIAGNOSTIC TESTER: Please enter a <CR> here: ";
```



```
my $a, $b = scalar <STDIN>;
print "\n";
print $x/$y;
```

If you prefer to run your program first and look at its problem afterwards, do this:

```
perl -w test.pl 2>test.out
./splain < test.out
```

Note that this is not in general possible in shells of more dubious heritage, as the theoretical

```
(perl -w test.pl >/dev/tty) >& test.out
./splain < test.out
```

Because you just moved the existing **stdout** to somewhere else.

If you don't want to modify your source code, but still have on-the-fly warnings, do this:

```
exec 3>&1; perl -w test.pl 2>&1 1>&3 3>&- | splain 1>&2 3>&-
```

Nifty, eh?

If you want to control warnings on the fly, do something like this. Make sure you do the use first, or you won't be able to get at the `enable()` or `disable()` methods.

```
use diagnostics; # checks entire compilation phase
print "\ntime for 1st bogus diags: SQUAWKINGS\n";
print BOGUS1 'nada';
print "done with 1st bogus\n";

disable diagnostics; # only turns off runtime warnings
print "\ntime for 2nd bogus: (squelched)\n";
print BOGUS2 'nada';
print "done with 2nd bogus\n";

enable diagnostics; # turns back on runtime warnings
print "\ntime for 3rd bogus: SQUAWKINGS\n";
print BOGUS3 'nada';
print "done with 3rd bogus\n";

disable diagnostics;
print "\ntime for 4th bogus: (squelched)\n";
print BOGUS4 'nada';
print "done with 4th bogus\n";
```

INTERNALS

Diagnostic messages derive from the *perl_{diag}.pod* file when available at runtime. Otherwise, they may be embedded in the file itself when the `splain` package is built. See the *Makefile* for details.

If an extant `$SIG{__WARN__}` handler is discovered, it will continue to be honored, but only after the `diagnostics::splainthis()` function (the module's `$SIG{__WARN__}` interceptor) has had its way with your warnings.

There is a `$diagnostics::DEBUG` variable you may set if you're desperately curious what sorts of things are being intercepted.

```
BEGIN { $diagnostics::DEBUG = 1 }
```

BUGS

Not being able to say "no diagnostics" is annoying, but may not be insurmountable.

The `-pretty` directive is called too late to affect matters. You have to do this instead, and *before* you load the module.

```
BEGIN { $diagnostics::PRETTY = 1 }
```

I could start up faster by delaying compilation until it should be needed, but this gets a "panic: top_level" when using the pragma form in Perl 5.001e.

While it's true that this documentation is somewhat subserious, if you use a program named *splain*, you should expect a bit of whimsy.

AUTHOR

Tom Christiansen <*tchrist@mox.perl.com*>, 25 June 1995.

NAME

Digest:: – Modules that calculate message digests

SYNOPSIS

```
$md2 = Digest->MD2;
$md5 = Digest->MD5;

$sha1 = Digest->SHA1;
$sha1 = Digest->new("SHA-1");

$hmac = Digest->HMAC_MD5($key);
```

DESCRIPTION

The `Digest::` modules calculate digests, also called "fingerprints" or "hashes", of some data, called a message. The digest is (usually) some small/fixed size string. The actual size of the digest depend of the algorithm used. The message is simply a sequence of arbitrary bytes.

An important property of the digest algorithms is that the digest is *likely* to change if the message change in some way. Another property is that digest functions are one-way functions, i.e. it should be *hard* to find a message that correspond to some given digest. Algorithms differ in how "likely" and how "hard", as well as how efficient they are to compute.

All `Digest::` modules provide the same programming interface. A functional interface for simple use, as well as an object oriented interface that can handle messages of arbitrary length and which can read files directly.

The digest can be delivered in three formats:

binary This is the most compact form, but it is not well suited for printing or embedding in places that can't handle arbitrary data.

hex A twice as long string of (lowercase) hexadecimal digits.

base64 A string of portable printable characters. This is the base64 encoded representation of the digest with any trailing padding removed. The string will be about 30% longer than the binary version. [MIME::Base64](#) tells you more about this encoding.

The functional interface is simply importable functions with the same name as the algorithm. The functions take the message as argument and return the digest. Example:

```
use Digest::MD5 qw(md5);
$digest = md5($message);
```

There are also versions of the functions with `"_hex"` or `"_base64"` appended to the name, which returns the digest in the indicated form.

OO INTERFACE

The following methods are available for all `Digest::` modules:

```
$ctx = Digest-XXX($arg, ...)
$ctx = Digest-new(XXX = $arg, ...)
$ctx = Digest::XXX-new($arg, ...)
```

The constructor returns some object that encapsulate the state of the message-digest algorithm. You can add data to the object and finally ask for the digest. The "XXX" should of course be replaced by the proper name of the digest algorithm you want to use.

The two first forms are simply syntactic sugar which automatically load the right module on first use. The second form allow you to use algorithm names which contains letters which are not legal perl identifiers, e.g. "SHA-1".

If `new()` is called as an instance method (i.e. `$ctx->new`) it will just reset the state the object to the state of a newly created object. No new object is created in this case, and the return value is the

reference to the object (i.e. `$ctx`) .

`$ctx-reset`

This is just an alias for `$ctx-new`.

`$ctx-add($data,...)`

The `$data` provided as argument are appended to the message we calculate the digest for. The return value is the `$ctx` object itself.

`$ctx-addfile($io_handle)`

The `$io_handle` is read until EOF and the content is appended to the message we calculate the digest for. The return value is the `$ctx` object itself.

`$ctx-digest`

Return the binary digest for the message.

Note that the `digest` operation is effectively a destructive, read-once operation. Once it has been performed, the `$ctx` object is automatically `reset` and can be used to calculate another digest value.

`$ctx-hexdigest`

Same as `$ctx-digest`, but will return the digest in hexadecimal form.

`$ctx-b64digest`

Same as `$ctx-digest`, but will return the digest as a base64 encoded string.

SEE ALSO

[Digest::MD5](#), [Digest::SHA1](#), [Digest::HMAC](#), [Digest::MD2](#)

[MIME::Base64](#)

AUTHOR

Gisle Aas <gisle@aa.no>

The `Digest::` interface is based on the interface originally developed by Neil Winton for his MD5 module.

NAME

Digest::MD5 – Perl interface to the MD5 Algorithm

SYNOPSIS

```
# Functional style
use Digest::MD5 qw(md5 md5_hex md5_base64);

$digest = md5($data);
$digest = md5_hex($data);
$digest = md5_base64($data);

# OO style
use Digest::MD5;

$ctx = Digest::MD5->new;

$ctx->add($data);
$ctx->addfile(*FILE);

$digest = $ctx->digest;
$digest = $ctx->hexdigest;
$digest = $ctx->b64digest;
```

DESCRIPTION

The Digest::MD5 module allows you to use the RSA Data Security Inc. MD5 Message Digest algorithm from within Perl programs. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input.

The Digest::MD5 module provide a procedural interface for simple use, as well as an object oriented interface that can handle messages of arbitrary length and which can read files directly.

A binary digest will be 16 bytes long. A hex digest will be 32 characters long. A base64 digest will be 22 characters long.

FUNCTIONS

The following functions can be exported from the Digest::MD5 module. No functions are exported by default.

`md5($data, ...)`

This function will concatenate all arguments, calculate the MD5 digest of this "message", and return it in binary form.

`md5_hex($data, ...)`

Same as `md5()`, but will return the digest in hexadecimal form.

`md5_base64($data, ...)`

Same as `md5()`, but will return the digest as a base64 encoded string.

The base64 encoded string returned is not padded to be a multiple of 4 bytes long. If you want interoperability with other base64 encoded md5 digests you might want to append the string "==" to the result.

METHODS

The following methods are available:

`$md5 = Digest::MD5->new`

The constructor returns a new Digest::MD5 object which encapsulate the state of the MD5 message-digest algorithm. You can add data to the object and finally ask for the digest.

If called as an instance method (i.e. `$md5->new`) it will just reset the state the object to the state of a newly created object. No new object is created in this case.

\$md5-reset

This is just an alias for `$md5-new`.

\$md5-add(\$data,...)

The `$data` provided as argument are appended to the message we calculate the digest for. The return value is the `$md5` object itself.

\$md5-addfile(\$io_handle)

The `$io_handle` is read until EOF and the content is appended to the message we calculate the digest for. The return value is the `$md5` object itself.

In most cases you want to make sure that the `$io_handle` is set up to be in `binmode()`.

\$md5-digest

Return the binary digest for the message.

Note that the `digest` operation is effectively a destructive, read-once operation. Once it has been performed, the `Digest::MD5` object is automatically reset and can be used to calculate another digest value.

\$md5-hexdigest

Same as `$md5-digest`, but will return the digest in hexadecimal form.

\$md5-b64digest

Same as `$md5-digest`, but will return the digest as a base64 encoded string.

The base64 encoded string returned is not padded to be a multiple of 4 bytes long. If you want interoperability with other base64 encoded md5 digests you might want to append the string `"=="` to the result.

EXAMPLES

The simplest way to use this library is to import the `md5_hex()` function (or one of its cousins):

```
use Digest::MD5 qw(md5_hex);
print "Digest is ", md5_hex("foobarbaz"), "\n";
```

The above example would print out the message

```
Digest is 6df23dc03f9b54cc38a0fc1483df6e21
```

provided that the implementation is working correctly. The same checksum can also be calculated in OO style:

```
use Digest::MD5;

$md5 = Digest::MD5->new;
$md5->add('foo', 'bar');
$md5->add('baz');
$digest = $md5->hexdigest;

print "Digest is $digest\n";
```

With OO style you can break the message arbitrary. This means that we are no longer limited to have space for the whole message in memory, i.e. we can handle messages of any size.

This is useful when calculating checksum for files:

```
use Digest::MD5;

my $file = shift || "/etc/passwd";
open(FILE, $file) or die "Can't open '$file': $!";
binmode(FILE);
```

```
$md5 = Digest::MD5->new;
while (<FILE>) {
    $md5->add($_);
}
close(FILE);
print $md5->b64digest, " $file\n";
```

Or we can use the builtin addfile method for more efficient reading of the file:

```
use Digest::MD5;

my $file = shift || "/etc/passwd";
open(FILE, $file) or die "Can't open '$file': $!";
binmode(FILE);

print Digest::MD5->new->addfile(*FILE)->hexdigest, " $file\n";
```

SEE ALSO

Digest, *Digest::MD2*, *Digest::SHA1*, *Digest::HMAC*

md5sum(1)

RFC 1321

COPYRIGHT

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Copyright 1998–2002 Gisle Aas.

Copyright 1995–1996 Neil Winton.

Copyright 1991–1992 RSA Data Security, Inc.

The MD5 algorithm is defined in RFC 1321. The basic C code implementing the algorithm is derived from that in the RFC and is covered by the following copyright:

Copyright (C) 1991–2, RSA Data Security, Inc. Created 1991. All rights reserved.

License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message–Digest Algorithm" in all material mentioning or referencing this software or this function.

License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message–Digest Algorithm" in all material mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this documentation and/or software.

This copyright does not prohibit distribution of any version of Perl containing this extension under the terms of the GNU or Artistic licenses.

AUTHORS

The original MD5 interface was written by Neil Winton (N.Winton@axion.bt.co.uk).

This release was made by Gisle Aas <gisle@ActiveState.com>

NAME

DirHandle – supply object methods for directory handles

SYNOPSIS

```
use DirHandle;
$d = new DirHandle ".";
if (defined $d) {
    while (defined($_ = $d->read)) { something($_); }
    $d->rewind;
    while (defined($_ = $d->read)) { something_else($_); }
    undef $d;
}
```

DESCRIPTION

The `DirHandle` method provide an alternative interface to the `opendir()`, `closedir()`, `readdir()`, and `rewinddir()` functions.

The only objective benefit to using `DirHandle` is that it avoids namespace pollution by creating globs to hold directory handles.

NOTES

- On Mac OS (Classic), the path separator is `':'`, not `'/'`, and the current directory is denoted as `':'`, not `'.'`. You should be careful about specifying relative pathnames. While a full path always begins with a volume name, a relative pathname should always begin with a `':'`. If specifying a volume name only, a trailing `':'` is required.

NAME

Dumpvalue – provides screen dump of Perl data.

SYNOPSIS

```
use Dumpvalue;
my $dumper = new Dumpvalue;
$dumper->set(globPrint => 1);
$dumper->dumpValue(\*::);
$dumper->dumpvars('main');
my $dump = $dumper->stringify($some_value);
```

DESCRIPTION**Creation**

A new dumper is created by a call

```
$d = new Dumpvalue(option1 => value1, option2 => value2)
```

Recognized options:

`arrayDepth`, `hashDepth`

Print only first N elements of arrays and hashes. If false, prints all the elements.

`compactDump`, `veryCompact`

Change style of array and hash dump. If true, short array may be printed on one line.

`globPrint`

Whether to print contents of globs.

`dumpDBFiles`

Dump arrays holding contents of debugged files.

`dumpPackages`

Dump symbol tables of packages.

`dumpReused`

Dump contents of "reused" addresses.

`tick`, `quoteHighBit`, `printUndef`

Change style of string dump. Default value of `tick` is `auto`, one can enable either double-quotish dump, or single-quotish by setting it to `"` or `'`. By default, characters with high bit set are printed *as is*. If `quoteHighBit` is set, they will be quoted.

`usageOnly`

rudimentally per-package memory usage dump. If set, `dumpvars` calculates total size of strings in variables in the package.

`unctrl`

Changes the style of printout of strings. Possible values are `unctrl` and `quote`.

`subdump`

Whether to try to find the subroutine name given the reference.

`bareStringify`

Whether to write the non-overloaded form of the `stringify`-overloaded objects.

`quoteHighBit`

Whether to print chars with high bit set in binary or "as is".

stopDbSignal

Whether to abort printing if debugger signal flag is raised.

Later in the life of the object the methods may be queried with `get()` method and `set()` method (which accept multiple arguments).

Methods**dumpValue**

```
$dumper->dumpValue($value);
$dumper->dumpValue([$value1, $value2]);
```

Prints a dump to the currently selected filehandle.

dumpValues

```
$dumper->dumpValues($value1, $value2);
```

Same as `$dumper->dumpValue([$value1, $value2]);`.

stringify

```
my $dump = $dumper->stringify($value [, $noticks] );
```

Returns the dump of a single scalar without printing. If the second argument is true, the return value does not contain enclosing ticks. Does not handle data structures.

dumpvars

```
$dumper->dumpvars('my_package');
$dumper->dumpvars('my_package', 'foo', '~bar$', '!.....');
```

The optional arguments are considered as literal strings unless they start with `~` or `!`, in which case they are interpreted as regular expressions (possibly negated).

The second example prints entries with names `foo`, and also entries with names which ends on `bar`, or are shorter than 5 chars.

set_quote

```
$d->set_quote('');
```

Sets `tick` and `unctrl` options to suitable values for printout with the given quote char. Possible values are `auto`, `'` and `"`.

set_unctrl

```
$d->set_unctrl('');
```

Sets `unctrl` option with checking for an invalid argument. Possible values are `unctrl` and `quote`.

compactDump

```
$d->compactDump(1);
```

Sets `compactDump` option. If the value is 1, sets to a reasonable big number.

veryCompact

```
$d->veryCompact(1);
```

Sets `compactDump` and `veryCompact` options simultaneously.

set

```
$d->set(option1 => value1, option2 => value2);
```

get

```
@values = $d->get('option1', 'option2');
```

NAME

Encode::Byte – Single Byte Encodings

SYNOPSIS

```
use Encode qw/encode decode/;
$greek = encode("iso-8859-7", $utf8); # loads Encode::Byte implicitly
$utf8  = decode("iso-8859-7", $greek); # ditto
```

ABSTRACT

This module implements various single byte encodings. For most cases it uses \x80–\xff (upper half) to map non-ASCII characters. Encodings supported are as follows.

Canonical	Alias	Description

# ISO 8859 series		
(iso-8859-1	is in built-in)	
iso-8859-2	latin2	[ISO]
iso-8859-3	latin3	[ISO]
iso-8859-4	latin4	[ISO]
iso-8859-5		[ISO]
iso-8859-6		[ISO]
iso-8859-7		[ISO]
iso-8859-8		[ISO]
iso-8859-9	latin5	[ISO]
iso-8859-10	latin6	[ISO]
iso-8859-11		
(iso-8859-12	is nonexistent)	
iso-8859-13	latin7	[ISO]
iso-8859-14	latin8	[ISO]
iso-8859-15	latin9	[ISO]
iso-8859-16	latin10	[ISO]
# Cyrillic		
koi8-f		
koi8-r	cp878	[RFC1489]
koi8-u		[RFC2319]
# Vietnamese		
viscii		
# all cp* are also available as ibm-*, ms-*, and windows-*		
# also see L< http://msdn.microsoft.com/workshop/author/dhtml/reference/charsets/cha		
cp424		
cp437		
cp737		
cp775		
cp850		
cp852		
cp855		
cp856		
cp857		
cp860		
cp861		
cp862		
cp863		
cp864		

```
cp865
cp866
cp869
cp874
cp1006
cp1250WinLatin2
cp1251WinCyrillic
cp1252WinLatin1
cp1253WinGreek
cp1254WinTurkish
cp1255WinHebrew
cp1256WinArabic
cp1257WinBaltic
cp1258WinVietnamese

# Macintosh
# Also see L<http://developer.apple.com/technotes/tn/tn1150.html>
MacArabic
MacCentralEurRoman
MacCroatian
MacCyrillic
MacFarsi
MacGreek
MacHebrew
MacIcelandic
MacRoman
MacRomanian
MacRumanian
MacSami
MacThai
MacTurkish
MacUkrainian

# More vendor encodings
AdobeStandardEncoding
nextstep
gsm0338      # used in GSM handsets
hp-roman8
```

DESCRIPTION

To find how to use this module in detail, see [Encode](#).

SEE ALSO

[Encode](#)

NAME

Encode::CN – China-based Chinese Encodings

SYNOPSIS

```
use Encode qw/encode decode/;
$euc_cn = encode("euc-cn", $utf8); # loads Encode::CN implicitly
$utf8   = decode("euc-cn", $euc_cn); # ditto
```

DESCRIPTION

This module implements China-based Chinese charset encodings. Encodings supported are as follows.

Canonical	Alias	Description

euc-cn	/\beuc.*cn\$/i /\bcn.*euc\$/i /\bGB[-_]?2312(?:\D.*\$ \$/i (see below)	EUC (Extended Unix Character)
gb2312-raw		The raw (low-bit) GB2312 character map
gb12345-raw		Traditional chinese counterpart to GB2312 (raw)
iso-ir-165		GB2312 + GB6345 + GB8565 + additions
MacChineseSimp		GB2312 + Apple Additions
cp936		Code Page 936, also known as GBK (Extended GuoBiao)
hz		7-bit escaped GB2312 encoding

To find how to use this module in detail, see [Encode](#).

NOTES

Due to size concerns, GB 18030 (an extension to GBK) is distributed separately on CPAN, under the name [Encode::HanExtra](#). That module also contains extra Taiwan-based encodings.

BUGS

When you see `charset=gb2312` on mails and web pages, they really mean `euc-cn` encodings. To fix that, `gb2312` is aliased to `euc-cn`. Use `gb2312-raw` when you really mean it.

The ASCII region (0x00–0x7f) is preserved for all encodings, even though this conflicts with mappings by the Unicode Consortium. See

<http://www.debian.or.jp/~kubota/unicode-symbols.html.en>

to find out why it is implemented that way.

SEE ALSO

[Encode](#)

NAME

Encode::EBCDIC – EBCDIC Encodings

SYNOPSIS

```
use Encode qw/encode decode/;
$posix_bc = encode("posix-bc", $utf8); # loads Encode::EBCDIC implicitly
$utf8 = decode("", $posix_bc);         # ditto
```

ABSTRACT

This module implements various EBCDIC-Based encodings. Encodings supported are as follows.

Canonical	Alias	Description
-----------	-------	-------------

cp37		
cp500		
cp875		
cp1026		
cp1047		
posix-bc		

DESCRIPTION

To find how to use this module in detail, see [Encode](#).

SEE ALSO

[Encode](#), [perlebcdic](#)

NAME

Encode::PerlIO — a detailed document on Encode and PerlIO

Overview

It is very common to want to do encoding transformations when reading or writing files, network connections, pipes etc. If Perl is configured to use the new 'perlio' IO system then Encode provides a "layer" (see [PerlIO](#)) which can transform data as it is read or written.

Here is how the blind poet would modernise the encoding:

```
use Encode;
open(my $iliad, '<:encoding(iso-8859-7)', 'iliad.greek');
open(my $utf8, '>:utf8', 'iliad.utf8');
my @epic = <$iliad>;
print $utf8 @epic;
close($utf8);
close($iliad);
```

In addition, the new IO system can also be configured to read/write UTF-8 encoded characters (as noted above, this is efficient):

```
open(my $fh, '>:utf8', 'anything');
print $fh "Any \x{0021} string \N{SMILEY FACE}\n";
```

Either of the above forms of "layer" specifications can be made the default for a lexical scope with the use open ... pragma. See [open](#).

Once a handle is open, its layers can be altered using binmode.

Without any such configuration, or if Perl itself is built using the system's own IO, then write operations assume that the file handle accepts only *bytes* and will die if a character larger than 255 is written to the handle. When reading, each octet from the handle becomes a byte-in-a-character. Note that this default is the same behaviour as bytes-only languages (including Perl before v5.6) would have, and is sufficient to handle native 8-bit encodings e.g. iso-8859-1, EBCDIC etc. and any legacy mechanisms for handling other encodings and binary data.

In other cases, it is the program's responsibility to transform characters into bytes using the API above before doing writes, and to transform the bytes read from a handle into characters before doing "character operations" (e.g. lc, /\W+/, ...).

You can also use PerlIO to convert larger amounts of data you don't want to bring into memory. For example, to convert between ISO-8859-1 (Latin 1) and UTF-8 (or UTF-EBCDIC in EBCDIC machines):

```
open(F, "<:encoding(iso-8859-1)", "data.txt") or die $!;
open(G, ">:utf8", "data.utf") or die $!;
while (<F>) { print G }

# Could also do "print G <F>" but that would pull
# the whole file into memory just to write it out again.
```

More examples:

```
open(my $f, "<:encoding(cp1252)")
open(my $g, ">:encoding(iso-8859-2)")
open(my $h, ">:encoding(latin9)") # iso-8859-15
```

See also [encoding](#) for how to change the default encoding of the data in your script.

How does it work?

Here is a crude diagram of how filehandle, PerlIO, and Encode interact.

How can I tell whether my encoding fully supports PerlIO ?

As of this writing, any encoding whose class belongs to `Encode::XS` and `Encode::Unicode` works. The `Encode` module has a `perlio_ok` method which you can use before applying PerlIO encoding to the filehandle. Here is an example:

```
my $use_perlio = perlio_ok($enc);
my $layer = $use_perlio ? "<:raw" : "<:encoding($enc)";
open my $fh, $layer, $file or die "$file : $!";
while(<$fh>){
    $_ = decode($enc, $_) unless $use_perlio;
    # ....
}
```

SEE ALSO

[Encode::Encoding](#), [Encode::Supported](#), [Encode::PerlIO](#), [encoding](#), [perlebcdic](#), [open](#), [perlunicode](#), [utf8](#), the Perl Unicode Mailing List <perl-unicode@perl.org>

NAME

Encode::Supported — Encodings supported by Encode

DESCRIPTION**Encoding Names**

Encoding names are case insensitive. White space in names is ignored. In addition, an encoding may have aliases. Each encoding has one "canonical" name. The "canonical" name is chosen from the names of the encoding by picking the first in the following sequence (with a few exceptions).

- The name used by the Perl community. That includes 'utf8' and 'ascii'. Unlike aliases, canonical names directly reach the method so such frequently used words like 'utf8' don't need to do alias lookups.
- The MIME name as defined in IETF RFCs. This includes all "iso-"s.
- The name in the IANA registry.
- The name used by the organization that defined it.

In case *de jure* canonical names differ from that of the Encode module, they are always aliased if it ever be implemented. So you can safely tell if a given encoding is implemented or not just by passing the canonical name.

Because of all the alias issues, and because in the general case encodings have state, "Encode" uses an encoding object internally once an operation is in progress.

Supported Encodings

As of Perl 5.8.0, at least the following encodings are recognized. Note that unless otherwise specified, they are all case insensitive (via alias) and all occurrence of spaces are replaced with '-'. In other words, "ISO 8859 1" and "iso-8859-1" are identical.

Encodings are categorized and implemented in several different modules but you don't have to use `Encode:::XX` to make them available for most cases. `Encode.pm` will automatically load those modules on demand.

Built-in Encodings

The following encodings are always available.

Canonical	Aliases	Comments & References
ascii	US-ascii ISO-646-US	[ECMA]
ascii-ctrl		Special Encoding
iso-8859-1	latin1	[ISO]
null		Special Encoding
utf8	UTF-8	[RFC2279]

null and *ascii-ctrl* are special. "null" fails for all character so when you set fallback mode to PERLQQ, HTMLCREF or XMLCREF, ALL CHARACTERS will fall back to character references. Ditto for "ascii-ctrl" except for control characters. For fallback modes, see [Encode](#).

Encode::Unicode — other Unicode encodings

Unicode coding schemes other than native utf8 are supported by `Encode::Unicode`, which will be autoloaded on demand.

UCS-2BE	UCS-2, iso-10646-1	[IANA, UC]
UCS-2LE		[UC]
UTF-16		[UC]
UTF-16BE		[UC]

UTF-16LE		[UC]
UTF-32		[UC]
UTF-32BE	UCS-4	[UC]
UTF-32LE		[UC]

To find how (UCS-2|UTF-(16|32))(LE|BE)? differ from one another, see [Encode::Unicode](#).

Encode::Byte — Extended ASCII

Encode::Byte implements most single-byte encodings except for Symbols and EBCDIC. The following encodings are based on single-byte encodings implemented as extended ASCII. Most of them map \x80–\xff (upper half) to non-ASCII characters.

ISO-8859 and corresponding vendor mappings

Since there are so many, they are presented in table format with languages and corresponding encoding names by vendors. Note that the table is sorted in order of ISO-8859 and the corresponding vendor mappings are slightly different from that of ISO. See <http://czyborra.com/charsets/iso8859.html> for details.

Lang/Regions	ISO/Other Std.	DOS	Windows	Macintosh	Others
N. America	(ASCII)	cp437		AdobeStandardEncoding	
		cp863	(DOSCanadaF)		
W. Europe	iso-8859-1	cp850	cp1252	MacRoman	nextstep hp-roman8
		cp860	(DOSPortuguese)		
Cntrl. Europe	iso-8859-2	cp852	cp1250	MacCentralEurRoman	
				MacCroatian	
				MacRomanian	
				MacRumanian	
Latin3 [1]	iso-8859-3				
Latin4 [2]	iso-8859-4				
Cyrillics	iso-8859-5	cp855	cp1251	MacCyrillic	
(See also next section)		cp866		MacUkrainian	
Arabic	iso-8859-6	cp864	cp1256	MacArabic	
		cp1006		MacFarsi	
Greek	iso-8859-7	cp737	cp1253	MacGreek	
		cp869	(DOSGreek2)		
Hebrew	iso-8859-8	cp862	cp1255	MacHebrew	
Turkish	iso-8859-9	cp857	cp1254	MacTurkish	
Nordics	iso-8859-10	cp865			
		cp861		MacIcelandic	
				MacSami	
Thai	iso-8859-11 [3]	cp874		MacThai	
(iso-8859-12 is nonexistent.	Reserved for Indics?)				
Baltics	iso-8859-13	cp775		cp1257	
Celtics	iso-8859-14				
Latin9 [4]	iso-8859-15				
Latin10	iso-8859-16				
Vietnamese	viscii		cp1258	MacVietnamese	

[1] Esperanto, Maltese, and Turkish. Turkish is now on 8859-9.

[2] Baltics. Now on 8859-10, except for Latvian.

[3] Also known as TIS 620.

[4] Nicknamed Latin0; the Euro sign as well as French and Finnish letters that are missing from 8859-1 were added.

All cp* are also available as ibm-*, ms-*, and windows-*. See also <http://czyborra.com/charsets/codepages.html>.

Macintosh encodings don't seem to be registered in such entities as IANA. "Canonical" names in Encode are based upon Apple's Tech Note 1150. See <http://developer.apple.com/technotes/tn/tn1150.html> for details.

KOI8 – De Facto Standard for the Cyrillic world

Though ISO-8859 does have ISO-8859-5, the KOI8 series is far more popular in the Net. *Encode* comes with the following KOI charsets. For gory details, see <http://czyborra.com/charsets/cyrillic.html>

```
-----
koi8-f
koi8-r cp878 [RFC1489]
koi8-u [RFC2319]
-----
```

gsm0338 – Hentai Latin 1

GSM0338 is for GSM handsets. Though it shares alphanumeric with ASCII, control character ranges and other parts are mapped very differently, presumably to store Greek and Cyrillic alphabets. This is also covered in Encode::Byte even though it is not an "extended ASCII" encoding.

CJK: Chinese, Japanese, Korean (Multibyte)

Note that Vietnamese is listed above. Also read "Encoding vs Charset" below. Also note that these are implemented in distinct modules by countries, due to the size concerns (simplified Chinese is mapped to 'CN', continental China, while traditional Chinese is mapped to 'TW', Taiwan). Please refer to their respective documentataion pages.

Encode::CN — Continental China

Standard	DOS/Win	Macintosh	Comment/Reference
euc-cn [1]		MacChineseSimp	
(gbk)	cp936 [2]		
gb12345-raw		{ GB12345 without CES }	
gb2312-raw		{ GB2312 without CES }	
hz			
iso-ir-165			

[1] GB2312 is aliased to this. See L<Microsoft-related naming mess>

[2] gbk is aliased to this. See L<Microsoft-related naming mess>

Encode::JP — Japan

Standard	DOS/Win	Macintosh	Comment/Reference
euc-jp			
shiftjis	cp932	macJapanese	
7bit-jis			
iso-2022-jp			[RFC1468]
iso-2022-jp-1			[RFC2237]
jis0201-raw	{ JIS X 0201 (roman + halfwidth kana)		without CES }
jis0208-raw	{ JIS X 0208 (Kanji + fullwidth kana)		without CES }
jis0212-raw	{ JIS X 0212 (Extended Kanji)		without CES }

Encode::KR — Korea

Standard	DOS/Win Macintosh	Comment/Reference
euc-kr	MacKorean	[RFC1557]
	cp949 [1]	
iso-2022-kr		[RFC1557]
johab		[KS X 1001:1998, Annex 3]
ksc5601-raw		{ KSC5601 without CES }

[1] ks_c_5601-1987, (x-)?windows-949, and uhc are aliased to this. See below.

Encode::TW — Taiwan

Standard	DOS/Win Macintosh	Comment/Reference
big5-eten	cp950 MacChineseTrad	{big5 aliased to big5-eten}
big5-hkscs		

Encode::HanExtra — More Chinese via CPAN

Due to size concerns, additional Chinese encodings below are distributed separately on CPAN, under the name Encode::HanExtra.

Standard	DOS/Win Macintosh	Comment/Reference
big5ext		CMEX's Big5e Extension
big5plus		CMEX's Big5+ Extension
cccii	Chinese Character Code for Information Interchange	
euc-tw		EUC (Extended Unix Character)
gb18030		GBK with Traditional Characters

Encode::JIS2K — JIS X 0213 encodings via CPAN

Due to size concerns, additional Japanese encodings below are distributed separately on CPAN, under the name Encode::JIS2K.

Standard	DOS/Win Macintosh	Comment/Reference
euc-jisx0213		
shiftjisx0123		
iso-2022-jp-3		
jis0213-1-raw		
jis0213-2-raw		

Miscellaneous encodings

Encode::EBCDIC

See [perlebcdic](#) for details.

```
-----
cp37
cp500
cp875
cp1026
cp1047
posix-bc
```

Encode::Symbols

For symbols and dingbats.

```
symbol
dingbats
MacDingbats
AdobeZdingbat
AdobeSymbol
```

Encode::MIME::Header

Strictly speaking, MIME header encoding documented in RFC 2047 is more of encapsulation than encoding. But included anyway.

MIME-Header	[RFC2047]
MIME-B	[RFC2047]
MIME-Q	[RFC2047]

Encode::Guess

This one is not a name of encoding but a utility that lets you pick up the most appropriate encoding for a data out of given *suspects*. See [Encode::Guess](#) for details.

Unsupported encodings

The following encodings are not supported as yet; some because they are rarely used, some because of technical difficulties. They may be supported by external modules via CPAN in the future, however.

ISO-2022-JP-2 [RFC1554]

Not very popular yet. Needs Unicode Database or equivalent to implement `encode()` (because it includes JIS X 0208/0212, KSC5601, and GB2312 simultaneously, whose code points in Unicode overlap. So you need to lookup the database to determine to what character set a given Unicode character should belong).

ISO-2022-CN [RFC1922]

Not very popular. Needs CNS 11643-1 and -2 which are not available in this module. CNS 11643 is supported (via `eutw`) in `Encode::HanExtra`. Atrijus Tang may add support for this encoding in his module in future.

Various HP-UX encodings

The following are unsupported due to the lack of mapping data.

```
'8' - arabic8, greek8, hebrew8, kana8, thai8, and turkish8
'15' - japanese15, korean15, and roi15
```

Cyrillic encoding ISO-IR-111

Anton Tagunov doubts its usefulness.

ISO-8859-8-1 [Hebrew]

None of the Encode team knows Hebrew enough (ISO-8859-8, `cp1255` and `MacHebrew` are supported because and just because there were mappings available at <http://www.unicode.org/>). Contributions welcome.

ISIRI 3342, Iran System, ISIRI 2900 [Farsi]

Ditto.

Thai encoding TCVN

Ditto.

Vietnamese encodings VPS

Though Jungshik Shin has reported that Mozilla supports this encoding, it was too late before 5.8.0 for us to add it. In the future, it may be available via a separate module. See

<http://lxr.mozilla.org/seamoney/source/intl/uconv/ucvlatin/vps.uf> and

<http://lxr.mozilla.org/seamoney/source/intl/uconv/ucvlatin/vps.ut> if you are interested in helping us.

Various Mac encodings

The following are unsupported due to the lack of mapping data.

MacArmenian, MacBengali, MacBurmese, MacEthiopic
MacExtArabic, MacGeorgian, MacKannada, MacKhmer
MacLaotian, MacMalayalam, MacMongolian, MacOriya
MacSinhalese, MacTamil, MacTelugu, MacTibetan
MacVietnamese

The rest which are already available are based upon the vendor mappings at

<http://www.unicode.org/Public/MAPPINGS/VENDORS/APPLE/>.

(Mac) Indic encodings

The maps for the following are available at <http://www.unicode.org/> but remain unsupported because those encodings need algorithmical approach, currently unsupported by *enc2xs*:

MacDevanagari
MacGurmukhi
MacGujarati

For details, please see Unicode mapping issues and notes: at

<http://www.unicode.org/Public/MAPPINGS/VENDORS/APPLE/DEVANAGA.TXT>.

I believe this issue is prevalent not only for Mac Indic but also in other Indic encodings, but the above were the only Indic encodings maps that I could find at <http://www.unicode.org/>.

Encoding vs. Charset — terminology

We are used to using the term (character) *encoding* and *character set* interchangeably. But just as confusing the terms byte and character is dangerous and the terms should be differentiated when needed, we need to differentiate *encoding* and *character set*.

To understand that, here is a description of how we make computers grok our characters.

- First we start with which characters to include. We call this collection of characters *character repertoire*.
- Then we have to give each character a unique ID so your computer can tell the difference between ‘a’ and ‘A’. This itemized character repertoire is now a *character set*.
- If your computer can grow the character set without further processing, you can go ahead and use it. This is called a *coded character set* (CCS) or *raw character encoding*. ASCII is used this way for most cases.
- But in many cases, especially multi-byte CJK encodings, you have to tweak a little more. Your network connection may not accept any data with the Most Significant Bit set, and your computer may not be able to tell if a given byte is a whole character or just half of it. So you have to *encode* the character set to use it.

A *character encoding scheme* (CES) determines how to encode a given character set, or a set of multiple character sets. 7bit ISO-2022 is an example of a CES. You switch between character sets via *escape sequences*.

Technically, or mathematically, speaking, a character set encoded in such a CES that maps character by character may form a CCS. EUC is such an example. The CES of EUC is as follows:

- Map ASCII unchanged.
- Map such a character set that consists of 94 or 96 powered by N members by adding 0x80 to each byte.
- You can also use 0x8e and 0x8f to indicate that the following sequence of characters belongs to yet another character set. To each following byte is added the value 0x80.

By carefully looking at the encoded byte sequence, you can find that the byte sequence conforms a unique number. In that sense, EUC is a CCS generated by a CES above from up to four CCS (complicated?). UTF-8 falls into this category. See [UTF-8 in perlUnicode](#) to find out how UTF-8 maps Unicode to a byte sequence.

You may also have found out by now why 7bit ISO-2022 cannot comprise a CCS. If you look at a byte sequence `\x21\x21`, you can't tell if it is two `!`'s or IDEOGRAPHIC SPACE. EUC maps the latter to `\xA1\xA1` so you have no trouble differentiating between `!!`, and `" "`.

Encoding Classification (by Anton Tagunov and Dan Kogai)

This section tries to classify the supported encodings by their applicability for information exchange over the Internet and to choose the most suitable aliases to name them in the context of such communication.

- To (en/de)code encodings marked by `(**)`, you need `Encode::HanExtra`, available from CPAN.

Encoding names

US-ASCII	UTF-8	ISO-8859-*	KOI8-R
Shift_JIS	EUC-JP	ISO-2022-JP	ISO-2022-JP-1
EUC-KR	Big5	GB2312	

are registered with IANA as preferred MIME names and may be used over the Internet.

Shift_JIS has been officialized by JIS X 0208:1997. [Microsoft-related naming mess](#) gives details.

GB2312 is the IANA name for EUC-CN. See [Microsoft-related naming mess](#) for details.

GB_2312-80 raw encoding is available as `gb2312-raw` with Encode. See [Encode::CN](#) for details.

EUC-CN	
KOI8-U	[RFC2319]

have not been registered with IANA (as of March 2002) but seem to be supported by major web browsers. The IANA name for EUC-CN is GB2312.

KS_C_5601-1987

is heavily misused. See [Microsoft-related naming mess](#) for details.

KS_C_5601-1987 raw encoding is available as `kcs5601-raw` with Encode. See [Encode::KR](#) for details.

UTF-16 UTF-16BE UTF-16LE

are IANA-registered charsets. See [RFC 2781] for details. Jungshik Shin reports that UTF-16 with a BOM is well accepted by MS IE 5/6 and NS 4/6. Beware however that

- UTF-16 support in any software you're going to be using/interoperating with has probably been less tested than UTF-8 support
- UTF-8 coded data seamlessly passes traditional command piping (`cat`, `more`, etc.) while UTF-16 coded data is likely to cause confusion (with its zero bytes, for example)

- it is beyond the power of words to describe the way HTML browsers encode non-ASCII form data. To get a general impression, visit <http://ppewww.ph.gla.ac.uk/~flavell/charset/form-i18n.html>. While encoding of form data has stabilized for UTF-8 encoded pages (at least IE 5/6, NS 6, and Opera 6 behave consistently), be sure to expect fun (and cross-browser discrepancies) with UTF-16 encoded pages!

The rule of thumb is to use UTF-8 unless you know what you're doing and unless you really benefit from using UTF-16.

```
ISO-IR-165      [RFC1345]
VISCII
GB 12345
GB 18030 (**)   (see links bellow)
EUC-TW         (**)
```

are totally valid encodings but not registered at IANA. The names under which they are listed here are probably the most widely-known names for these encodings and are recommended names.

```
BIG5PLUS (**)

```

is a proprietary name.

Microsoft-related naming mess

Microsoft products misuse the following names:

KS_C_5601-1987

Microsoft extension to EUC-KR.

Proper names: CP949, UHC, x-windows-949 (as used by Mozilla).

See <http://lists.w3.org/Archives/Public/ietf-charsets/2001AprJun/0033.html> for details.

Encode aliases KS_C_5601-1987 to cp949 to reflect this common misuse. *Raw* KS_C_5601-1987 encoding is available as kcs5601-raw.

See [Encode::KR](#) for details.

GB2312

Microsoft extension to EUC-CN.

Proper names: CP936, GBK.

GB2312 has been registered in the EUC-CN meaning at IANA. This has partially repaired the situation: Microsoft's GB2312 has become a superset of the official GB2312.

Encode aliases GB2312 to euc-cn in full agreement with IANA registration. cp936 is supported separately. *Raw* GB_2312-80 encoding is available as gb2312-raw.

See [Encode::CN](#) for details.

Big5

Microsoft extension to Big5.

Proper name: CP950.

Encode separately supports Big5 and cp950.

Shift_JIS

Microsoft's understanding of Shift_JIS.

JIS has not endorsed the full Microsoft standard however. The official Shift_JIS includes only JIS X 0201 and JIS X 0208 character sets, while Microsoft has always used Shift_JIS to encode a wider character repertoire. See IANA registration for Windows-31J.

As a historical predecessor, Microsoft's variant probably has more rights for the name, though it may be objected that Microsoft shouldn't have used JIS as part of the name in the first place.

Unambiguous name: CP932. IANA name (not used?): Windows-31J.

Encode separately supports Shift_JIS and cp932.

Glossary

character repertoire

A collection of unique characters. A *character* set in the strictest sense. At this stage, characters are not numbered.

coded character set (CCS)

A character set that is mapped in a way computers can use directly. Many character encodings, including EUC, fall in this category.

character encoding scheme (CES)

An algorithm to map a character set to a byte sequence. You don't have to be able to tell which character set a given byte sequence belongs. 7-bit ISO-2022 is a CES but it cannot be a CCS. EUC is an example of being both a CCS and CES.

charset (in MIME context)

has long been used in the meaning of encoding, CES.

While the word combination `character set` has lost this meaning in MIME context since [RFC 2130], the `charset` abbreviation has retained it. This is how [RFC 2277] and [RFC 2278] bless `charset`:

This document uses the term "charset" to mean a set of rules for mapping from a sequence of octets to a sequence of characters, such as the combination of a coded character set and a character encoding scheme; this is also what is used as an identifier in MIME "charset=" parameters, and registered in the IANA charset registry ... (Note that this is NOT a term used by other standards bodies, such as ISO).
[RFC 2277]

EUC

Extended Unix Character. See ISO-2022.

ISO-2022

A CES that was carefully designed to coexist with ASCII. There are a 7 bit version and an 8 bit version.

The 7 bit version switches character set via escape sequence so it cannot form a CCS. Since this is more difficult to handle in programs than the 8 bit version, the 7 bit version is not very popular except for iso-2022-jp, the *de facto* standard CES for e-mails.

The 8 bit version can form a CCS. EUC and ISO-8859 are two examples thereof. Pre-5.6 perl could use them as string literals.

UCS

Short for *Universal Character Set*. When you say just UCS, it means *Unicode*.

UCS-2

ISO/IEC 10646 encoding form: Universal Character Set coded in two octets.

Unicode

A character set that aims to include all character repertoires of the world. Many character sets in various national as well as industrial standards have become, in a way, just subsets of Unicode.

UTF

Short for *Unicode Transformation Format*. Determines how to map a Unicode character into a byte sequence.

UTF-16

A UTF in 16-bit encoding. Can either be in big endian or little endian. The big endian version is called UTF-16BE (equal to UCS-2 + surrogate support) and the little endian version is called UTF-16LE.

See Also

[Encode](#), [Encode::Byte](#), [Encode::CN](#), [Encode::JP](#), [Encode::KR](#), [Encode::TW](#), [Encode::EBCDIC](#), [Encode::Symbol](#), [Encode::MIME::Header](#), [Encode::Guess](#)

References

ECMA

European Computer Manufacturers Association <http://www.ecma.ch>

ECMA-035 (eq ISO-2022)

<http://www.ecma.ch/ecma1/STAND/ECMA-035.HTM>

The specification of ISO-2022 is available from the link above.

IANA

Internet Assigned Numbers Authority <http://www.iana.org/>

Assigned Charset Names by IANA

<http://www.iana.org/assignments/character-sets>

Most of the `canonical` names in `Encode` derive from this list so you can directly apply the string you have extracted from MIME header of mails and web pages.

ISO International Organization for Standardization <http://www.iso.ch/>

RFC

Request For Comments — need I say more? <http://www.rfc-editor.org/>, <http://www.rfc.net/>, <http://www.faqs.org/rfcs/>

UC Unicode Consortium <http://www.unicode.org/>

Unicode Glossary

<http://www.unicode.org/glossary/>

The glossary of this document is based upon this site.

Other Notable Sites

czyborra.com

<http://czyborra.com/>

Contains a a lot of useful information, especially gory details of ISO vs. vendor mappings.

CJK.inf

http://www.oreilly.com/people/authors/lunde/cjk_inf.html

Somewhat obsolete (last update in 1996), but still useful. Also try

ftp://ftp.oreilly.com/pub/examples/nutshell/cjk/pdf/GB18030_Summary.pdf

You will find brief info on EUC-CN, GBK and mostly on GB 18030.

Jungshik Shin's Hangul FAQ

<http://jshin.net/faq>

And especially its subject 8.

<http://jshin.net/faq/qa8.html>

A comprehensive overview of the Korean (KS *) standards.

debian.org: "Introduction to i18n"

A brief description for most of the mentioned CJK encodings is contained in

<http://www.debian.org/doc/manuals/intro-i18n/ch-codes.en.html>

Offline sources

CJKV Information Processing by Ken Lunde

CJKV Information Processing 1999 O'Reilly & Associates, ISBN : 1-56592-224-7

The modern successor of CJK.inf.

Features a comprehensive coverage of CJKV character sets and encodings along with many other issues faced by anyone trying to better support CJKV languages/scripts in all the areas of information processing.

To purchase this book, visit <http://www.oreilly.com/catalog/cjkvinfo/> or your favourite bookstore.

NAME

Encode – character encodings

SYNOPSIS

```
use Encode;
```

Table of Contents

Encode consists of a collection of modules whose details are too big to fit in one document. This POD itself explains the top-level APIs and general topics at a glance. For other topics and more details, see the PODs below:

Name	Description
Encode::Alias	Alias definitions to encodings
Encode::Encoding	Encode Implementation Base Class
Encode::Supported	List of Supported Encodings
Encode::CN	Simplified Chinese Encodings
Encode::JP	Japanese Encodings
Encode::KR	Korean Encodings
Encode::TW	Traditional Chinese Encodings

DESCRIPTION

The Encode module provides the interfaces between Perl's strings and the rest of the system. Perl strings are sequences of **characters**.

The repertoire of characters that Perl can represent is at least that defined by the Unicode Consortium. On most platforms the ordinal values of the characters (as returned by `ord(ch)`) is the "Unicode codepoint" for the character (the exceptions are those platforms where the legacy encoding is some variant of EBCDIC rather than a super-set of ASCII – see [perlebcdic](#)).

Traditionally, computer data has been moved around in 8-bit chunks often called "bytes". These chunks are also known as "octets" in networking standards. Perl is widely used to manipulate data of many types – not only strings of characters representing human or computer languages but also "binary" data being the machine's representation of numbers, pixels in an image – or just about anything.

When Perl is processing "binary data", the programmer wants Perl to process "sequences of bytes". This is not a problem for Perl – as a byte has 256 possible values, it easily fits in Perl's much larger "logical character".

TERMINOLOGY

- *character*: a character in the range 0..(2**32–1) (or more). (What Perl's strings are made of.)
- *byte*: a character in the range 0..255 (A special case of a Perl character.)
- *octet*: 8 bits of data, with ordinal values 0..255 (Term for bytes passed to or from a non-Perl context, e.g. a disk file.)

PERL ENCODING API

```
$octets = encode(ENCODING, $string [, CHECK])
```

Encodes a string from Perl's internal form into *ENCODING* and returns a sequence of octets. ENCODING can be either a canonical name or an alias. For encoding names and aliases, see [. For CHECK, see \[.\]\(#\)](#)

For example, to convert a string from Perl's internal format to iso-8859-1 (also known as Latin1),

```
$octets = encode("iso-8859-1", $string);
```

CAVEAT: When you run `$octets = encode("utf8", $string)`, then `$octets` **may not be**

equal to `$string`. Though they both contain the same data, the utf8 flag for `$octets` is **always** off. When you encode anything, utf8 flag of the result is always off, even when it contains completely valid utf8 string. See below.

`encode($valid_encoding, undef)` is harmless but warns you for Use of uninitialized value in subroutine entry. `encode($valid_encoding, "")` is harmless and warnless.

`$string = decode(ENCODING, $octets [, CHECK])`

Decodes a sequence of octets assumed to be in *ENCODING* into Perl's internal form and returns the resulting string. As in `encode()`, *ENCODING* can be either a canonical name or an alias. For encoding names and aliases, see . For *CHECK*, see .

For example, to convert ISO-8859-1 data to a string in Perl's internal format:

```
$string = decode("iso-8859-1", $octets);
```

CAVEAT: When you run `$string = decode("utf8", $octets)`, then `$string` **may not be equal to** `$octets`. Though they both contain the same data, the utf8 flag for `$string` is on unless `$octets` entirely consists of ASCII data (or EBCDIC on EBCDIC machines). See below.

`decode($valid_encoding, undef)` is harmless but warns you for Use of uninitialized value in subroutine entry. `decode($valid_encoding, "")` is harmless and warnless.

`[$length =] from_to($octets, FROM_ENC, TO_ENC [, CHECK])`

Converts **in-place** data between two encodings. The data in `$octets` must be encoded as octets and not as characters in Perl's internal format. For example, to convert ISO-8859-1 data to Microsoft's CP1250 encoding:

```
from_to($octets, "iso-8859-1", "cp1250");
```

and to convert it back:

```
from_to($octets, "cp1250", "iso-8859-1");
```

Note that because the conversion happens in place, the data to be converted cannot be a string constant; it must be a scalar variable.

`from_to()` returns the length of the converted string in octets on success, `undef` otherwise.

CAVEAT: The following operations look the same but are not quite so;

```
from_to($data, "iso-8859-1", "utf8"); #1
$data = decode("iso-8859-1", $data); #2
```

Both #1 and #2 make `$data` consist of a completely valid UTF-8 string but only #2 turns utf8 flag on. #1 is equivalent to

```
$data = encode("utf8", decode("iso-8859-1", $data));
```

See below.

`$octets = encode_utf8($string);`

Equivalent to `$octets = encode("utf8", $string)`; The characters that comprise `$string` are encoded in Perl's internal format and the result is returned as a sequence of octets. All possible characters have a UTF-8 representation so this function cannot fail.

`$string = decode_utf8($octets [, CHECK]);`

equivalent to `$string = decode("utf8", $octets [, CHECK])`. The sequence of octets represented by `$octets` is decoded from UTF-8 into a sequence of logical characters. Not all sequences of octets form valid UTF-8 encodings, so it is possible for this call to fail. For *CHECK*, see .

Listing available encodings

```
use Encode;
@list = Encode->encodings();
```

Returns a list of the canonical names of the available encodings that are loaded. To get a list of all available encodings including the ones that are not loaded yet, say

```
@all_encodings = Encode->encodings(":all");
```

Or you can give the name of a specific module.

```
@with_jp = Encode->encodings("Encode::JP");
```

When ":" is not in the name, "Encode:" is assumed.

```
@ebcdic = Encode->encodings("EBCDIC");
```

To find out in detail which encodings are supported by this package, see [Encode::Supported](#).

Defining Aliases

To add a new alias to a given encoding, use:

```
use Encode;
use Encode::Alias;
define_alias(newName => ENCODING);
```

After that, `newName` can be used as an alias for `ENCODING`. `ENCODING` may be either the name of an encoding or an *encoding object*

But before you do so, make sure the alias is nonexistent with `resolve_alias()`, which returns the canonical name thereof. i.e.

```
Encode::resolve_alias("latin1") eq "iso-8859-1" # true
Encode::resolve_alias("iso-8859-12")           # false; nonexistent
Encode::resolve_alias($name) eq $name          # true if $name is canonical
```

`resolve_alias()` does not need `use Encode::Alias`; it can be exported via `use Encode qw(resolve_alias)`.

See [Encode::Alias](#) for details.

Encoding via PerlIO

If your perl supports *PerlIO* (which is the default), you can use a PerlIO layer to decode and encode directly via a filehandle. The following two examples are totally identical in their functionality.

```
# via PerlIO
open my $in, "<:encoding(shiftjis)", $infile or die;
open my $out, ">:encoding(euc-jp)", $outfile or die;
while(<$in){ print $out $_; }

# via from_to
open my $in, "<", $infile or die;
open my $out, ">", $outfile or die;
while(<$in){
    from_to($_, "shiftjis", "euc-jp", 1);
    print $out $_;
}
```

Unfortunately, it may be that encodings are PerlIO-savvy. You can check if your encoding is supported by PerlIO by calling the `perlio_ok` method.

```
Encode::perlio_ok("hz");           # False
find_encoding("euc-cn")->perlio_ok; # True where PerlIO is available
```

```
use Encode qw(perlio_ok);          # exported upon request
perlio_ok("euc-jp")
```

Fortunately, all encodings that come with Encode core are PerlIO-savvy except for hz and ISO-2022-kr. For gory details, see [Encode::Encoding](#) and [Encode::PerlIO](#).

Handling Malformed Data

The *CHECK* argument is used as follows. When you omit it, the behaviour is the same as if you had passed a value of 0 for *CHECK*.

CHECK = Encode::FB_DEFAULT (== 0)

If *CHECK* is 0, (en|de)code will put a *substitution character* in place of a malformed character. For UCM-based encodings, <subchar> will be used. For Unicode, the code point 0xFFFD is used. If the data is supposed to be UTF-8, an optional lexical warning (category utf8) is given.

CHECK = Encode::FB_CROAK (== 1)

If *CHECK* is 1, methods will die on error immediately with an error message. Therefore, when *CHECK* is set to 1, you should trap the fatal error with eval{ } unless you really want to let it die on error.

CHECK = Encode::FB_QUIET

If *CHECK* is set to Encode::FB_QUIET, (en|de)code will immediately return the portion of the data that has been processed so far when an error occurs. The data argument will be overwritten with everything after that point (that is, the unprocessed part of data). This is handy when you have to call decode repeatedly in the case where your source data may contain partial multi-byte character sequences, for example because you are reading with a fixed-width buffer. Here is some sample code that does exactly this:

```
my $data = ''; my $utf8 = '';
while(defined(read $fh, $buffer, 256)){
    # buffer may end in a partial character so we append
    $data .= $buffer;
    $utf8 .= decode($encoding, $data, Encode::FB_QUIET);
    # $data now contains the unprocessed partial character
}
```

CHECK = Encode::FB_WARN

This is the same as above, except that it warns on error. Handy when you are debugging the mode above.

perlqq mode (*CHECK* = Encode::FB_PERLQQ)

HTML charref mode (*CHECK* = Encode::FB_HTMLCREF)

XML charref mode (*CHECK* = Encode::FB_XMLCREF)

For encodings that are implemented by Encode::XS, *CHECK* == Encode::FB_PERLQQ turns (en|de)code into perlqq fallback mode.

When you decode, \xHH will be inserted for a malformed character, where HH is the hex representation of the octet that could not be decoded to utf8. And when you encode, \x{HHHH} will be inserted, where HHHH is the Unicode ID of the character that cannot be found in the character repertoire of the encoding.

HTML/XML character reference modes are about the same, in place of \x{HHHH}, HTML uses &#NNNN; where NNNN is a decimal digit and XML uses &#xHHHH; where HHHH is the hexadecimal digit.

The bitmask

These modes are actually set via a bitmask. Here is how the FB_XX constants are laid out. You can import the FB_XX constants via use Encode qw(:fallbacks); you can import the generic bitmask constants via use Encode qw(:fallback_all).

```
FB_DEFAULT FB_CROAK FB_QUIET FB_WARN FB_PERLQQ
```


DIE_ON_ERR	0x0001	X	
WARN_ON_ERR	0x0002		X
RETURN_ON_ERR	0x0004	X	X
LEAVE_SRC	0x0008		
PERLQQ	0x0100		X
HTMLCREF	0x0200		
XMLCREF	0x0400		

Unimplemented fallback schemes

In the future, you will be able to use a code reference to a callback function for the value of *CHECK* but its API is still undecided.

The fallback scheme does not work on EBCDIC platforms.

Defining Encodings

To define a new encoding, use:

```
use Encode qw(define_encoding);
define_encoding($object, 'canonicalName' [, alias...]);
```

canonicalName will be associated with *\$object*. The object should provide the interface described in [Encode::Encoding](#). If more than two arguments are provided then additional arguments are taken as aliases for *\$object*.

See [Encode::Encoding](#) for more details.

The UTF-8 flag

Before the introduction of utf8 support in perl, The eq operator just compared the strings represented by two scalars. Beginning with perl 5.8, eq compares two strings with simultaneous consideration of *the utf8 flag*. To explain why we made it so, I will quote page 402 of *Programming Perl*, 3rd ed.

Goal #1:

Old byte-oriented programs should not spontaneously break on the old byte-oriented data they used to work on.

Goal #2:

Old byte-oriented programs should magically start working on the new character-oriented data when appropriate.

Goal #3:

Programs should run just as fast in the new character-oriented mode as in the old byte-oriented mode.

Goal #4:

Perl should remain one language, rather than forking into a byte-oriented Perl and a character-oriented Perl.

Back when *Programming Perl*, 3rd ed. was written, not even Perl 5.6.0 was born and many features documented in the book remained unimplemented for a long time. Perl 5.8 corrected this and the introduction of the UTF-8 flag is one of them. You can think of this perl notion as of a byte-oriented mode (utf8 flag off) and a character-oriented mode (utf8 flag on).

Here is how Encode takes care of the utf8 flag.

- When you encode, the resulting utf8 flag is always off.

When you decode, the resulting utf8 flag is on unless you can unambiguously represent data. Here is the definition of dis-ambiguity.

```
After $utf8 = decode('foo', $octet);,
```

```
When $octet is... The utf8 flag in $utf8 is
```

```
-----
```

In ASCII only (or EBCDIC only)	OFF
In ISO-8859-1	ON
In any other Encoding	ON

As you see, there is one exception, In ASCII. That way you can assue Goal #1. And with Encode Goal #2 is assumed but you still have to be careful in such cases mentioned in **CAVEAT** paragraphs.

This utf8 flag is not visible in perl scripts, exactly for the same reason you cannot (or you *don't have to*) see if a scalar contains a string, integer, or floating point number. But you can still peek and poke these if you will. See the section below.

Messing with Perl's Internals

The following API uses parts of Perl's internals in the current implementation. As such, they are efficient but may change.

`is_utf8(String [, CHECK])`

[INTERNAL] Tests whether the UTF-8 flag is turned on in the STRING. If CHECK is true, also checks the data in STRING for being well-formed UTF-8. Returns true if successful, false otherwise.

`_utf8_on(String)`

[INTERNAL] Turns on the UTF-8 flag in STRING. The data in STRING is **not** checked for being well-formed UTF-8. Do not use unless you **know** that the STRING is well-formed UTF-8. Returns the previous state of the UTF-8 flag (so please don't treat the return value as indicating success or failure), or undef if STRING is not a string.

`_utf8_off(String)`

[INTERNAL] Turns off the UTF-8 flag in STRING. Do not use frivolously. Returns the previous state of the UTF-8 flag (so please don't treat the return value as indicating success or failure), or undef if STRING is not a string.

SEE ALSO

[Encode::Encoding](#), [Encode::Supported](#), [Encode::PerlIO](#), [encoding](#), [perlebcdic](#), [open](#), [perlunicode](#), [utf8](#), the Perl Unicode Mailing List <perl-unicode@perl.org>

MAINTAINER

This project was originated by Nick Ing-Simmons and later maintained by Dan Kogai <dankogai@dan.co.jp>. See AUTHORS for a full list of people involved. For any questions, use <perl-unicode@perl.org> so we can all share.

NAME

encoding – allows you to write your script in non-ascii or non-utf8

SYNOPSIS

```
use encoding "greek"; # Perl like Greek to you?
use encoding "euc-jp"; # Jperl!

# or you can even do this if your shell supports your native encoding

perl -Mencoding=latin2 -e '...' # Feeling centrally European?
perl -Mencoding=euc-kr -e '...' # Or Korean?

# more control

# A simple euc-cn => utf-8 converter
use encoding "euc-cn", STDOUT => "utf8"; while(<>){print};

# "no encoding;" supported (but not scoped!)
no encoding;

# an alternate way, Filter
use encoding "euc-jp", Filter=>1;
use utf8;
# now you can use kanji identifiers -- in euc-jp!
```

ABSTRACT

Let's start with a bit of history: Perl 5.6.0 introduced Unicode support. You could apply `substr()` and regexes even to complex CJK characters — so long as the script was written in UTF-8. But back then, text editors that supported UTF-8 were still rare and many users instead chose to write scripts in legacy encodings, giving up a whole new feature of Perl 5.6.

Rewind to the future: starting from perl 5.8.0 with the **encoding** pragma, you can write your script in any encoding you like (so long as the Encode module supports it) and still enjoy Unicode support. You can write code in EUC-JP as follows:

```
my $Rakuda = "\xF1\xD1\xF1\xCC"; # Camel in Kanji
           #<-char-><-char->      # 4 octets
s/\bCamel\b/$Rakuda/;
```

And with `use encoding "euc-jp"` in effect, it is the same thing as the code in UTF-8:

```
my $Rakuda = "\x{99F1}\x{99DD}"; # two Unicode Characters
s/\bCamel\b/$Rakuda/;
```

The **encoding** pragma also modifies the filehandle disciplines of STDIN, STDOUT, and STDERR to the specified encoding. Therefore,

```
use encoding "euc-jp";
my $message = "Camel is the symbol of perl.\n";
my $Rakuda = "\xF1\xD1\xF1\xCC"; # Camel in Kanji
$message =~ s/\bCamel\b/$Rakuda/;
print $message;
```

Will print `"\xF1\xD1\xF1\xCC is the symbol of perl.\n"`, not `"\x{99F1}\x{99DD} is the symbol of perl.\n"`.

You can override this by giving extra arguments; see below.

USAGE

`use encoding [ENCNAME];`

Sets the script encoding to *ENCNAME*. Filehandle disciplines of STDIN and STDOUT are set to `":encoding(ENCNAME)"`. Note that STDERR will not be changed.

If no encoding is specified, the environment variable *PERL_ENCODING* is consulted. If no encoding can be found, the error `Unknown encoding 'ENCNAME'` will be thrown.

Note that non-STD file handles remain unaffected. Use `use open` or `binmode` to change disciplines of those.

`use encoding ENCNAME [STDIN => ENCNAME_IN ...] ;`

You can also individually set encodings of STDIN and STDOUT via the `< STDIN = ENCNAME` form. In this case, you cannot omit the first *ENCNAME*. `< STDIN = undef` turns the IO transcoding completely off.

`no encoding;`

Unsets the script encoding. The disciplines of STDIN, STDOUT are reset to `":raw"` (the default unprocessed raw stream of bytes).

CAVEATS

NOT SCOPED

The pragma is a per script, not a per block lexical. Only the last `use encoding` or `no encoding` matters, and it affects **the whole script**. However, the `<no encoding` pragma is supported and **use encoding** can appear as many times as you want in a given script. The multiple use of this pragma is discouraged.

Because of this nature, the use of this pragma inside the module is strongly discouraged (because the influence of this pragma lasts not only for the module but the script that uses). But if you have to, make sure you say `no encoding` at the end of the module so you contain the influence of the pragma within the module.

DO NOT MIX MULTIPLE ENCODINGS

Notice that only literals (string or regular expression) having only legacy code points are affected: if you mix data like this

```
\xDF\x{100}
```

the data is assumed to be in (Latin 1 and) Unicode, not in your native encoding. In other words, this will match in "greek":

```
"\xDF" =~ /\x{3af}/
```

but this will not

```
"\xDF\x{100}" =~ /\x{3af}\x{100}/
```

since the `\xDF` (ISO 8859-7 GREEK SMALL LETTER IOTA WITH TONOS) on the left will **not** be upgraded to `\x{3af}` (Unicode GREEK SMALL LETTER IOTA WITH TONOS) because of the `\x{100}` on the left. You should not be mixing your legacy data and Unicode in the same string.

This pragma also affects encoding of the 0x80..0xFF code point range: normally characters in that range are left as eight-bit bytes (unless they are combined with characters with code points 0x100 or larger, in which case all characters need to become UTF-8 encoded), but if the `encoding` pragma is present, even the 0x80..0xFF range always gets UTF-8 encoded.

After all, the best thing about this pragma is that you don't have to resort to `\x{....}` just to spell your name in a native encoding. So feel free to put your strings in your encoding in quotes and regexes.

Non-ASCII Identifiers and Filter option

The magic of `use encoding` is not applied to the names of identifiers. In order to make `${"\x{4eba}"}++` (`$human++`, where `human` is a single Han ideograph) work, you still need to write your script in UTF-8 or use a source filter.

In other words, the same restriction as with `Jperl` applies.

If you dare to experiment, however, you can try the `Filter` option.

use encoding *ENCNAME* **Filter=>1**;

This turns the encoding pragma into a source filter. While the default approach just decodes interpolated literals (in `qq()` and `qr()`), this will apply a source filter to the entire source code. In this case, `STDIN` and `STDOUT` remain untouched.

What does this mean? Your source code behaves as if it is written in UTF-8. So even if your editor only supports Shift_JIS, for example, you can still try examples in Chapter 15 of *Programming Perl*, 3rd Ed.. For instance, you can use UTF-8 identifiers.

This option is significantly slower and (as of this writing) non-ASCII identifiers are not very stable WITHOUT this option and with the source code written in UTF-8.

To make your script in legacy encoding work with minimum effort, do not use `Filter=>1`.

EXAMPLE – Greekperl

```
use encoding "iso 8859-7";

# \xDF in ISO 8859-7 (Greek) is \x{3af} in Unicode.

$a = "\xDF";
$b = "\x{100}";

printf "%#x\n", ord($a); # will print 0x3af, not 0xdf

$c = $a . $b;

# $c will be "\x{3af}\x{100}", not "\x{df}\x{100}".
# chr() is affected, and ...

print "mega\n" if ord(chr(0xdf)) == 0x3af;

# ... ord() is affected by the encoding pragma ...

print "tera\n" if ord(pack("C", 0xdf)) == 0x3af;

# ... as are eq and cmp ...

print "peta\n" if "\x{3af}" eq pack("C", 0xdf);
print "exa\n" if "\x{3af}" cmp pack("C", 0xdf) == 0;

# ... but pack/unpack C are not affected, in case you still
# want to go back to your native encoding

print "zetta\n" if unpack("C", (pack("C", 0xdf))) == 0xdf;
```

KNOWN PROBLEMS

For native multibyte encodings (either fixed or variable length), the current implementation of the regular expressions may introduce recoding errors for regular expression literals longer than 127 bytes.

The encoding pragma is not supported on EBCDIC platforms. (Porters who are willing and able to remove this limitation are welcome.)

SEE ALSO

[perlunicode](#), [Encode](#), [open](#), [Filter::Util::Call](#),

Ch. 15 of *Programming Perl* (3rd Edition) by Larry Wall, Tom Christiansen, Jon Orwant; O'Reilly & Associates; ISBN 0-596-00027-8

NAME

Encode::JP – Japanese Encodings

SYNOPSIS

```
use Encode qw/encode decode/;
$euc_jp = encode("euc-jp", $utf8); # loads Encode::JP implicitly
$utf8   = decode("euc-jp", $euc_jp); # ditto
```

ABSTRACT

This module implements Japanese charset encodings. Encodings supported are as follows.

Canonical	Alias	Description
euc-jp	/\beuc.*jp\$/i /\bjp.*euc/i /\bujis\$/i	EUC (Extended Unix Character)
shiftjis	/\bshift.*jis\$/i /\bsjis\$/i	Shift JIS (aka MS Kanji)
7bit-jis	/\bjis\$/i	7bit JIS
iso-2022-jp		ISO-2022-JP [RFC1468] = 7bit JIS with all Halfwidth Kana converted to Fullwidth
iso-2022-jp-1		ISO-2022-JP-1 [RFC2237] = ISO-2022-JP with JIS X 0212-1990 support. See below
MacJapanese cp932		Shift JIS + Apple vendor mappings Code Page 932 = Shift JIS + MS/IBM vendor mappings
jis0201-raw		JIS0201, raw format
jis0208-raw		JIS0201, raw format
jis0212-raw		JIS0201, raw format

DESCRIPTION

To find out how to use this module in detail, see [Encode](#).

Note on ISO-2022-JP(-1)?

ISO-2022-JP-1 (RFC2237) is a superset of ISO-2022-JP (RFC1468) which adds support for JIS X 0212-1990. That means you can use the same code to decode to utf8 but not vice versa.

```
$utf8 = decode('iso-2022-jp-1', $stream);
```

and

```
$utf8 = decode('iso-2022-jp', $stream);
```

yield the same result but

```
$with_0212 = encode('iso-2022-jp-1', $utf8);
```

is now different from

```
$without_0212 = encode('iso-2022-jp', $utf8 );
```

In the latter case, characters that map to 0212 are first converted to U+3013 (0xA2AE in EUC-JP; a white square also known as ‘Tofu’ or ‘geta mark’) then fed to the decoding engine. U+FFFD is not used, in order to preserve text layout as much as possible.

BUGS

The ASCII region (0x00–0x7f) is preserved for all encodings, even though this conflicts with mappings by the Unicode Consortium. See

<http://www.debian.or.jp/~kubota/unicode-symbols.html.en>

to find out why it is implemented that way.

SEE ALSO

Encode

NAME

Encode::KR – Korean Encodings

SYNOPSIS

```
use Encode qw/encode decode/;
$euc_kr = encode("euc-kr", $utf8); # loads Encode::KR implicitly
$utf8   = decode("euc-kr", $euc_kr); # ditto
```

DESCRIPTION

This module implements Korean charset encodings. Encodings supported are as follows.

Canonical	Alias	Description
euc-kr	/\beuc.*kr\$/i /\bkr.*euc\$/i	EUC (Extended Unix Character)
ksc5601-raw		Korean standard code set (as is)
cp949	/(?:x-)?uhc\$/i /(?:x-)?windows-949\$/i /\bks_c_5601-1987\$/i	Code Page 949 (EUC-KR + 8,822 (additional Hangul syllables)
MacKorean		EUC-KR + Apple Vendor Mappings
johab	JOHAB	A supplementary encoding defined in Annex 3 of KS X 1001:1998
iso-2022-kr		iso-2022-kr [RFC1557]

To find how to use this module in detail, see [Encode](#).

BUGS

When you see charset=ks_c_5601-1987 on mails and web pages, they really mean "cp949" encodings. To fix that, the following aliases are set;

```
qr/(?:x-)?uhc$/i      => ' "cp949" '
qr/(?:x-)?windows-949$/i => ' "cp949" '
qr/ks_c_5601-1987$/i  => ' "cp949" '
```

The ASCII region (0x00–0x7f) is preserved for all encodings, even though this conflicts with mappings by the Unicode Consortium. See

<http://www.debian.or.jp/~kubota/unicode-symbols.html.en>

to find out why it is implemented that way.

SEE ALSO

[Encode](#)

NAME

Encode::Alias – alias definitions to encodings

SYNOPSIS

```
use Encode;
use Encode::Alias;
define_alias( newName => ENCODING );
```

DESCRIPTION

Allows *newName* to be used as an alias for *ENCODING*. *ENCODING* may be either the name of an encoding or an encoding object (as described in [Encode](#)).

Currently *newName* can be specified in the following ways:

As a simple string.

As a `qr//` compiled regular expression, e.g.:

```
define_alias( qr/^iso8859-(\d+)$/i => "iso-8859-$1" );
```

In this case, if *ENCODING* is not a reference, it is `eval`-ed in order to allow `$1` etc. to be substituted.

The example is one way to alias names as used in X11 fonts to the MIME names for the iso-8859-* family. Note the double quotes inside the single quotes.

If you are using a regex here, you have to use the quotes as shown or it won't work. Also note that regex handling is tricky even for the experienced. Use it with caution.

As a code reference, e.g.:

```
define_alias( sub { return /^iso8859-(\d+)$/i ? "iso-8859-$1" : undef } , '' );
```

In this case, `$_` will be set to the name that is being looked up and *ENCODING* is passed to the sub as its first argument. The example is another way to alias names as used in X11 fonts to the MIME names for the iso-8859-* family.

Alias overloading

You can override predefined aliases by simply applying `define_alias()`. The new alias is always evaluated first, and when necessary, `define_alias()` flushes the internal cache to make the new definition available.

```
# redirect SHIFT_JIS to MS/IBM Code Page 932, which is a
# superset of SHIFT_JIS

define_alias( qr/shift.*jis$/i   => "cp932" );
define_alias( qr/sjis$/i         => "cp932" );
```

If you want to zap all predefined aliases, you can use

```
Encode::Alias->undef_aliases;
```

to do so. And

```
Encode::Alias->init_aliases;
```

gets the factory settings back.

SEE ALSO

[Encode](#), [Encode::Supported](#)

NAME

Encode::CJKConstants.pm -- Internally used by Encode::??::ISO_2022_*

NAME

Encode::CN::HZ — internally used by Encode::CN

NAME

Encode::Config — internally used by Encode

NAME

Encode::Encoder — Object Oriented Encoder

SYNOPSIS

```
use Encode::Encoder;
# Encode::encode("ISO-8859-1", $data);
Encode::Encoder->new($data)->iso_8859_1; # OOP way
# shortcut
use Encode::Encoder qw(encoder);
encoder($data)->iso_8859_1;
# you can stack them!
encoder($data)->iso_8859_1->base64; # provided base64() is defined
# you can use it as a decoder as well
encoder($base64)->bytes('base64')->latin1;
# stringified
print encoder($data)->utf8->latin1; # prints the string in latin1
# numified
encoder("\x{abcd}\x{ef}g")->utf8 == 6; # true. bytes::length($data)
```

ABSTRACT

Encode::Encoder allows you to use Encode in an object-oriented style. This is not only more intuitive than a functional approach, but also handier when you want to stack encodings. Suppose you want your UTF-8 string converted to Latin1 then Base64: you can simply say

```
my $base64 = encoder($utf8)->latin1->base64;
```

instead of

```
my $latin1 = encode("latin1", $utf8);
my $base64 = encode_base64($utf8);
```

or the lazier and more convoluted

```
my $base64 = encode_base64(encode("latin1", $utf8));
```

Description

Here is how to use this module.

- There are at least two instance variables stored in a hash reference, {data} and {encoding}.
- When there is no method, it takes the method name as the name of the encoding and encodes the instance *data* with *encoding*. If successful, the instance *encoding* is set accordingly.
- You can retrieve the result via `->data` but usually you don't have to because the stringify operator (`""`) is overridden to do exactly that.

Predefined Methods

This module predefines the methods below:

```
$e = Encode::Encoder->new([$data, $encoding]);
```

returns an encoder object. Its data is initialized with *\$data* if present, and its encoding is set to *\$encoding* if present.

When *\$encoding* is omitted, it defaults to `utf8` if *\$data* is already in `utf8` or `""` (empty string) otherwise.

```
encoder()
```

is an alias of `Encode::Encoder->new()`. This one is exported on demand.

`$e->data([$data])`

When `$data` is present, sets the instance data to `$data` and returns the object itself. Otherwise, the current instance data is returned.

`$e->encoding([$encoding])`

When `$encoding` is present, sets the instance encoding to `$encoding` and returns the object itself. Otherwise, the current instance encoding is returned.

`$e->bytes([$encoding])`

decodes instance data from `$encoding`, or the instance encoding if omitted. If the conversion is successful, the instance encoding will be set to `""`.

The name *bytes* was deliberately picked to avoid namespace tainting — this module may be used as a base class so method names that appear in `Encode::Encoding` are avoided.

Example: base64 transcoder

This module is designed to work with [Encode::Encoding](#). To make the Base64 transcoder example above really work, you could write a module like this:

```
package Encode::Base64;
use base 'Encode::Encoding';
__PACKAGE__->Define('base64');
use MIME::Base64;
sub encode{
    my ($obj, $data) = @_;
    return encode_base64($data);
}
sub decode{
    my ($obj, $data) = @_;
    return decode_base64($data);
}
1;
__END__
```

And your caller module would be something like this:

```
use Encode::Encoder;
use Encode::Base64;

# now you can really do the following

encoder($data)->iso_8859_1->base64;
encoder($base64)->bytes('base64')->latin1;
```

Operator Overloading

This module overloads two operators, `stringify ("")` and `numify (0+)`.

`Stringify` dumps the data inside the object.

`Numify` returns the number of bytes in the instance data.

They come in handy when you want to print or find the size of data.

SEE ALSO

[Encode](#), [Encode::Encoding](#)

NAME

Encode::Encoding – Encode Implementation Base Class

SYNOPSIS

```
package Encode::MyEncoding;
use base qw(Encode::Encoding);

__PACKAGE__->Define(qw(myCanonical myAlias));
```

DESCRIPTION

As mentioned in [Encode](#), encodings are (in the current implementation at least) defined as objects. The mapping of encoding name to object is via the `%Encode::Encoding` hash. Though you can directly manipulate this hash, it is strongly encouraged to use this base class module and add `encode()` and `decode()` methods.

Methods you should implement

You are strongly encouraged to implement methods below, at least either `encode()` or `decode()`.

`->encode($string[, $check])`

MUST return the octet sequence representing *\$string*.

- If *\$check* is true, it SHOULD modify *\$string* in place to remove the converted part (i.e. the whole string unless there is an error). If `perlio_ok()` is true, SHOULD becomes MUST.
- If an error occurs, it SHOULD return the octet sequence for the fragment of string that has been converted and modify *\$string* in-place to remove the converted part leaving it starting with the problem fragment. If `perlio_ok()` is true, SHOULD becomes MUST.
- If *\$check* is false then `encode` MUST make a "best effort" to convert the string – for example, by using a replacement character.

`->decode($octets[, $check])`

MUST return the string that *\$octets* represents.

- If *\$check* is true, it SHOULD modify *\$octets* in place to remove the converted part (i.e. the whole sequence unless there is an error). If `perlio_ok()` is true, SHOULD becomes MUST.
- If an error occurs, it SHOULD return the fragment of string that has been converted and modify *\$octets* in-place to remove the converted part leaving it starting with the problem fragment. If `perlio_ok()` is true, SHOULD becomes MUST.
- If *\$check* is false then `decode` should make a "best effort" to convert the string – for example by using Unicode's `"\x{FFFD}"` as a replacement character.

Other methods defined in Encode::Encodings

You do not have to override methods shown below unless you have to.

`->name`

Predefined As:

```
sub name { return shift->{'Name'} }
```

MUST return the string representing the canonical name of the encoding.

`->new_sequence`

Predefined As:

```
sub new_sequence { return $_[0] }
```

This is a placeholder for encodings with state. It should return an object which implements this interface. All current implementations return the original object.

`->perlio_ok()`

Predefined As:

```
sub perlio_ok {
    eval{ require PerlIO::encoding };
    return $@ ? 0 : 1;
}
```

If your encoding does not support PerlIO for some reasons, just;

```
sub perlio_ok { 0 }
```

`->needs_lines()`

Predefined As:

```
sub needs_lines { 0 };
```

If your encoding can work with PerlIO but needs line buffering, you **MUST** define this method so it returns true. 7bit ISO-2022 encodings are one example that needs this. When this method is missing, false is assumed.

Example: Encode::ROT13

```
package Encode::ROT13;
use strict;
use base qw(Encode::Encoding);
__PACKAGE__->Define('rot13');

sub encode($$;$){
    my ($obj, $str, $chk) = @_;
    $str =~ tr/A-Za-z/N-ZA-Mn-za-m/;
    $_[1] = '' if $chk; # this is what in-place edit means
    return $str;
}

# Jr pna or ynm1 yvxr guvf;
*decode = \&encode;

1;
```

Why the heck Encode API is different?

It should be noted that the *\$check* behaviour is different from the outer public API. The logic is that the "unchecked" case is useful when the encoding is part of a stream which may be reporting errors (e.g. STDERR). In such cases, it is desirable to get everything through somehow without causing additional errors which obscure the original one. Also, the encoding is best placed to know what the correct replacement character is, so if that is the desired behaviour then letting low level code do it is the most efficient.

By contrast, if *\$check* is true, the scheme above allows the encoding to do as much as it can and tell the layer above how much that was. What is lacking at present is a mechanism to report what went wrong. The most likely interface will be an additional method call to the object, or perhaps (to avoid forcing per-stream objects on otherwise stateless encodings) an additional parameter.

It is also highly desirable that encoding classes inherit from `Encode::Encoding` as a base class. This allows that class to define additional behaviour for all encoding objects.

```
package Encode::MyEncoding;
use base qw(Encode::Encoding);
__PACKAGE__->Define(qw(myCanonical myAlias));
```


to create an object with `< bless {Name = ...}, $class`, and call `define_encoding`. They inherit their name method from `Encode::Encoding`.

Compiled Encodings

For the sake of speed and efficiency, most of the encodings are now supported via a *compiled form*: XS modules generated from UCM files. Encode provides the `enc2xs` tool to achieve that. Please see [enc2xs](#) for more details.

SEE ALSO

[perlmod](#), [enc2xs](#)

=begin future

Scheme 1

The fixup routine gets passed the remaining fragment of string being processed. It modifies it in place to remove bytes/characters it can understand and returns a string used to represent them. For example:

```
sub fixup {
    my $ch = substr($_[0], 0, 1, '');
    return sprintf("\x{%02X}", ord($ch);
}
```

This scheme is close to how the underlying C code for Encode works, but gives the fixup routine very little context.

Scheme 2

The fixup routine gets passed the original string, an index into it of the problem area, and the output string so far. It appends what it wants to the output string and returns a new index into the original string. For example:

```
sub fixup {
    # my ($s,$i,$d) = @_;
    my $ch = substr($_[0], $_[1], 1);
    $_[2] .= sprintf("\x{%02X}", ord($ch);
    return $_[1]+1;
}
```

This scheme gives maximal control to the fixup routine but is more complicated to code, and may require that the internals of Encode be tweaked to keep the original string intact.

Other Schemes

Hybrids of the above.

Multiple return values rather than in-place modifications.

Index into the string could be `pos($str)` allowing `s/\G...//`.

=end future

NAME

Encode::JP::H2Z — internally used by Encode::JP::2022_JP*

NAME

Encode::JP::JIS7 — internally used by Encode::JP

NAME

Encode::KR::2022_KR — internally used by Encode::KR

NAME

Encode::Supported — Encodings supported by Encode

DESCRIPTION**Encoding Names**

Encoding names are case insensitive. White space in names is ignored. In addition, an encoding may have aliases. Each encoding has one "canonical" name. The "canonical" name is chosen from the names of the encoding by picking the first in the following sequence (with a few exceptions).

- The name used by the Perl community. That includes 'utf8' and 'ascii'. Unlike aliases, canonical names directly reach the method so such frequently used words like 'utf8' don't need to do alias lookups.
- The MIME name as defined in IETF RFCs. This includes all "iso-"s.
- The name in the IANA registry.
- The name used by the organization that defined it.

In case *de jure* canonical names differ from that of the Encode module, they are always aliased if it ever be implemented. So you can safely tell if a given encoding is implemented or not just by passing the canonical name.

Because of all the alias issues, and because in the general case encodings have state, "Encode" uses an encoding object internally once an operation is in progress.

Supported Encodings

As of Perl 5.8.0, at least the following encodings are recognized. Note that unless otherwise specified, they are all case insensitive (via alias) and all occurrence of spaces are replaced with '-'. In other words, "ISO 8859 1" and "iso-8859-1" are identical.

Encodings are categorized and implemented in several different modules but you don't have to use `Encode:::XX` to make them available for most cases. `Encode.pm` will automatically load those modules on demand.

Built-in Encodings

The following encodings are always available.

Canonical	Aliases	Comments & References
ascii	US-ascii ISO-646-US	[ECMA]
ascii-ctrl		Special Encoding
iso-8859-1	latin1	[ISO]
null		Special Encoding
utf8	UTF-8	[RFC2279]

null and *ascii-ctrl* are special. "null" fails for all character so when you set fallback mode to PERLQQ, HTMLCREF or XMLCREF, ALL CHARACTERS will fall back to character references. Ditto for "ascii-ctrl" except for control characters. For fallback modes, see [Encode](#).

Encode::Unicode — other Unicode encodings

Unicode coding schemes other than native utf8 are supported by `Encode::Unicode`, which will be autoloaded on demand.

UCS-2BE	UCS-2, iso-10646-1	[IANA, UC]
UCS-2LE		[UC]
UTF-16		[UC]
UTF-16BE		[UC]

UTF-16LE		[UC]
UTF-32		[UC]
UTF-32BE	UCS-4	[UC]
UTF-32LE		[UC]

To find how (UCS-2|UTF-(16|32))(LE|BE)? differ from one another, see [Encode::Unicode](#).

Encode::Byte — Extended ASCII

Encode::Byte implements most single-byte encodings except for Symbols and EBCDIC. The following encodings are based on single-byte encodings implemented as extended ASCII. Most of them map \x80-\xff (upper half) to non-ASCII characters.

ISO-8859 and corresponding vendor mappings

Since there are so many, they are presented in table format with languages and corresponding encoding names by vendors. Note that the table is sorted in order of ISO-8859 and the corresponding vendor mappings are slightly different from that of ISO. See <http://czyborra.com/charsets/iso8859.html> for details.

Lang/Regions	ISO/Other Std.	DOS	Windows	Macintosh	Others
N. America	(ASCII)	cp437		AdobeStandardEncoding	
		cp863	(DOSCanadaF)		
W. Europe	iso-8859-1	cp850	cp1252	MacRoman	nextstep hp-roman8
		cp860	(DOSPortuguese)		
Cntrl. Europe	iso-8859-2	cp852	cp1250	MacCentralEurRoman	
				MacCroatian	
				MacRomanian	
				MacRumanian	
Latin3 [1]	iso-8859-3				
Latin4 [2]	iso-8859-4				
Cyrillics	iso-8859-5	cp855	cp1251	MacCyrillic	
(See also next section)		cp866		MacUkrainian	
Arabic	iso-8859-6	cp864	cp1256	MacArabic	
		cp1006		MacFarsi	
Greek	iso-8859-7	cp737	cp1253	MacGreek	
		cp869	(DOSGreek2)		
Hebrew	iso-8859-8	cp862	cp1255	MacHebrew	
Turkish	iso-8859-9	cp857	cp1254	MacTurkish	
Nordics	iso-8859-10	cp865			
		cp861		MacIcelandic	
				MacSami	
Thai	iso-8859-11 [3]	cp874		MacThai	
(iso-8859-12 is nonexistent.	Reserved for Indics?)				
Baltics	iso-8859-13	cp775		cp1257	
Celtics	iso-8859-14				
Latin9 [4]	iso-8859-15				
Latin10	iso-8859-16				
Vietnamese	viscii		cp1258	MacVietnamese	

[1] Esperanto, Maltese, and Turkish. Turkish is now on 8859-9.

[2] Baltics. Now on 8859-10, except for Latvian.

[3] Also known as TIS 620.

[4] Nicknamed Latin0; the Euro sign as well as French and Finnish letters that are missing from 8859-1 were added.

All cp* are also available as ibm-*, ms-*, and windows-*. See also <http://czyborra.com/charsets/codepages.html>.

Macintosh encodings don't seem to be registered in such entities as IANA. "Canonical" names in Encode are based upon Apple's Tech Note 1150. See <http://developer.apple.com/technotes/tn/tn1150.html> for details.

KOI8 – De Facto Standard for the Cyrillic world

Though ISO-8859 does have ISO-8859-5, the KOI8 series is far more popular in the Net. [Encode](#) comes with the following KOI charsets. For gory details, see <http://czyborra.com/charsets/cyrillic.html>

```
-----
koi8-f
koi8-r cp878 [RFC1489]
koi8-u [RFC2319]
-----
```

gsm0338 – Hentai Latin 1

GSM0338 is for GSM handsets. Though it shares alphanumeric with ASCII, control character ranges and other parts are mapped very differently, presumably to store Greek and Cyrillic alphabets. This is also covered in Encode::Byte even though it is not an "extended ASCII" encoding.

CJK: Chinese, Japanese, Korean (Multibyte)

Note that Vietnamese is listed above. Also read "Encoding vs Charset" below. Also note that these are implemented in distinct modules by countries, due to the size concerns (simplified Chinese is mapped to 'CN', continental China, while traditional Chinese is mapped to 'TW', Taiwan). Please refer to their respective documentataion pages.

Encode::CN — Continental China

Standard	DOS/Win	Macintosh	Comment/Reference
euc-cn [1]		MacChineseSimp	
(gbk)	cp936 [2]		
gb12345-raw		{ GB12345 without CES }	
gb2312-raw		{ GB2312 without CES }	
hz			
iso-ir-165			

[1] GB2312 is aliased to this. See L<Microsoft-related naming mess>

[2] gbk is aliased to this. See L<Microsoft-related naming mess>

Encode::JP — Japan

Standard	DOS/Win	Macintosh	Comment/Reference
euc-jp			
shiftjis	cp932	macJapanese	
7bit-jis			
iso-2022-jp			[RFC1468]
iso-2022-jp-1			[RFC2237]
jis0201-raw	{ JIS X 0201 (roman + halfwidth kana)		without CES }
jis0208-raw	{ JIS X 0208 (Kanji + fullwidth kana)		without CES }
jis0212-raw	{ JIS X 0212 (Extended Kanji)		without CES }

Encode::KR — Korea

Standard	DOS/Win Macintosh	Comment/Reference
euc-kr	MacKorean	[RFC1557]
	cp949 [1]	
iso-2022-kr		[RFC1557]
johab		[KS X 1001:1998, Annex 3]
ksc5601-raw		{ KSC5601 without CES }

[1] ks_c_5601-1987, (x-)?windows-949, and uhc are aliased to this. See below.

Encode::TW — Taiwan

Standard	DOS/Win Macintosh	Comment/Reference
big5-eten	cp950 MacChineseTrad	{big5 aliased to big5-eten}
big5-hkscs		

Encode::HanExtra — More Chinese via CPAN

Due to size concerns, additional Chinese encodings below are distributed separately on CPAN, under the name Encode::HanExtra.

Standard	DOS/Win Macintosh	Comment/Reference
big5ext		CMEX's Big5e Extension
big5plus		CMEX's Big5+ Extension
cccii	Chinese Character Code for Information Interchange	
euc-tw		EUC (Extended Unix Character)
gb18030		GBK with Traditional Characters

Encode::JIS2K — JIS X 0213 encodings via CPAN

Due to size concerns, additional Japanese encodings below are distributed separately on CPAN, under the name Encode::JIS2K.

Standard	DOS/Win Macintosh	Comment/Reference
euc-jisx0213		
shiftjisx0123		
iso-2022-jp-3		
jis0213-1-raw		
jis0213-2-raw		

Miscellaneous encodings

Encode::EBCDIC

See [perlebcdic](#) for details.

```
-----
cp37
cp500
cp875
cp1026
cp1047
posix-bc
```

Encode::Symbols

For symbols and dingbats.

```
symbol
dingbats
MacDingbats
AdobeZdingbat
AdobeSymbol
```

Encode::MIME::Header

Strictly speaking, MIME header encoding documented in RFC 2047 is more of encapsulation than encoding. But included anyway.

MIME-Header	[RFC2047]
MIME-B	[RFC2047]
MIME-Q	[RFC2047]

Encode::Guess

This one is not a name of encoding but a utility that lets you pick up the most appropriate encoding for a data out of given *suspects*. See [Encode::Guess](#) for details.

Unsupported encodings

The following encodings are not supported as yet; some because they are rarely used, some because of technical difficulties. They may be supported by external modules via CPAN in the future, however.

ISO-2022-JP-2 [RFC1554]

Not very popular yet. Needs Unicode Database or equivalent to implement `encode()` (because it includes JIS X 0208/0212, KSC5601, and GB2312 simultaneously, whose code points in Unicode overlap. So you need to lookup the database to determine to what character set a given Unicode character should belong).

ISO-2022-CN [RFC1922]

Not very popular. Needs CNS 11643-1 and -2 which are not available in this module. CNS 11643 is supported (via `euc-tw`) in `Encode::HanExtra`. Atrijus Tang may add support for this encoding in his module in future.

Various HP-UX encodings

The following are unsupported due to the lack of mapping data.

```
'8' - arabic8, greek8, hebrew8, kana8, thai8, and turkish8
'15' - japanese15, korean15, and roi15
```

Cyrillic encoding ISO-IR-111

Anton Tagunov doubts its usefulness.

ISO-8859-8-1 [Hebrew]

None of the Encode team knows Hebrew enough (ISO-8859-8, `cp1255` and `MacHebrew` are supported because and just because there were mappings available at <http://www.unicode.org/>). Contributions welcome.

ISIRI 3342, Iran System, ISIRI 2900 [Farsi]

Ditto.

Thai encoding TCVN

Ditto.

Vietnamese encodings VPS

Though Jungshik Shin has reported that Mozilla supports this encoding, it was too late before 5.8.0 for us to add it. In the future, it may be available via a separate module. See

<http://lxr.mozilla.org/seamoney/source/intl/uconv/ucvlatin/vps.uf> and

<http://lxr.mozilla.org/seamoney/source/intl/uconv/ucvlatin/vps.ut> if you are interested in helping us.

Various Mac encodings

The following are unsupported due to the lack of mapping data.

MacArmenian, MacBengali, MacBurmese, MacEthiopic
MacExtArabic, MacGeorgian, MacKannada, MacKhmer
MacLaotian, MacMalayalam, MacMongolian, MacOriya
MacSinhalese, MacTamil, MacTelugu, MacTibetan
MacVietnamese

The rest which are already available are based upon the vendor mappings at

<http://www.unicode.org/Public/MAPPINGS/VENDORS/APPLE/>.

(Mac) Indic encodings

The maps for the following are available at <http://www.unicode.org/> but remain unsupported because those encodings need algorithmical approach, currently unsupported by *enc2xs*:

MacDevanagari
MacGurmukhi
MacGujarati

For details, please see Unicode mapping issues and notes: at

<http://www.unicode.org/Public/MAPPINGS/VENDORS/APPLE/DEVANAGA.TXT>.

I believe this issue is prevalent not only for Mac Indic but also in other Indic encodings, but the above were the only Indic encodings maps that I could find at <http://www.unicode.org/>.

Encoding vs. Charset — terminology

We are used to using the term (character) *encoding* and *character set* interchangeably. But just as confusing the terms byte and character is dangerous and the terms should be differentiated when needed, we need to differentiate *encoding* and *character set*.

To understand that, here is a description of how we make computers grok our characters.

- First we start with which characters to include. We call this collection of characters *character repertoire*.
- Then we have to give each character a unique ID so your computer can tell the difference between ‘a’ and ‘A’. This itemized character repertoire is now a *character set*.
- If your computer can grow the character set without further processing, you can go ahead and use it. This is called a *coded character set* (CCS) or *raw character encoding*. ASCII is used this way for most cases.
- But in many cases, especially multi-byte CJK encodings, you have to tweak a little more. Your network connection may not accept any data with the Most Significant Bit set, and your computer may not be able to tell if a given byte is a whole character or just half of it. So you have to *encode* the character set to use it.

A *character encoding scheme* (CES) determines how to encode a given character set, or a set of multiple character sets. 7bit ISO-2022 is an example of a CES. You switch between character sets via *escape sequences*.

Technically, or mathematically, speaking, a character set encoded in such a CES that maps character by character may form a CCS. EUC is such an example. The CES of EUC is as follows:

- Map ASCII unchanged.
- Map such a character set that consists of 94 or 96 powered by N members by adding 0x80 to each byte.
- You can also use 0x8e and 0x8f to indicate that the following sequence of characters belongs to yet another character set. To each following byte is added the value 0x80.

By carefully looking at the encoded byte sequence, you can find that the byte sequence conforms a unique number. In that sense, EUC is a CCS generated by a CES above from up to four CCS (complicated?). UTF-8 falls into this category. See [UTF-8 in perlUnicode](#) to find out how UTF-8 maps Unicode to a byte sequence.

You may also have found out by now why 7bit ISO-2022 cannot comprise a CCS. If you look at a byte sequence `\x21\x21`, you can't tell if it is two `!`'s or IDEOGRAPHIC SPACE. EUC maps the latter to `\xA1\xA1` so you have no trouble differentiating between `!!` and `" "`.

Encoding Classification (by Anton Tagunov and Dan Kogai)

This section tries to classify the supported encodings by their applicability for information exchange over the Internet and to choose the most suitable aliases to name them in the context of such communication.

- To (en/de)code encodings marked by `(**)`, you need `Encode::HanExtra`, available from CPAN.

Encoding names

US-ASCII	UTF-8	ISO-8859-*	KOI8-R
Shift_JIS	EUC-JP	ISO-2022-JP	ISO-2022-JP-1
EUC-KR	Big5	GB2312	

are registered with IANA as preferred MIME names and may be used over the Internet.

Shift_JIS has been officialized by JIS X 0208:1997. [Microsoft-related naming mess](#) gives details.

GB2312 is the IANA name for EUC-CN. See [Microsoft-related naming mess](#) for details.

GB_2312-80 raw encoding is available as `gb2312-raw` with Encode. See [Encode::CN](#) for details.

EUC-CN	
KOI8-U	[RFC2319]

have not been registered with IANA (as of March 2002) but seem to be supported by major web browsers. The IANA name for EUC-CN is GB2312.

KS_C_5601-1987

is heavily misused. See [Microsoft-related naming mess](#) for details.

KS_C_5601-1987 raw encoding is available as `kcs5601-raw` with Encode. See [Encode::KR](#) for details.

UTF-16 UTF-16BE UTF-16LE

are IANA-registered charsets. See [RFC 2781] for details. Jungshik Shin reports that UTF-16 with a BOM is well accepted by MS IE 5/6 and NS 4/6. Beware however that

- UTF-16 support in any software you're going to be using/interoperating with has probably been less tested than UTF-8 support
- UTF-8 coded data seamlessly passes traditional command piping (`cat`, `more`, etc.) while UTF-16 coded data is likely to cause confusion (with its zero bytes, for example)

- it is beyond the power of words to describe the way HTML browsers encode non-ASCII form data. To get a general impression, visit <http://ppewww.ph.gla.ac.uk/~flavell/charset/form-i18n.html>. While encoding of form data has stabilized for UTF-8 encoded pages (at least IE 5/6, NS 6, and Opera 6 behave consistently), be sure to expect fun (and cross-browser discrepancies) with UTF-16 encoded pages!

The rule of thumb is to use UTF-8 unless you know what you're doing and unless you really benefit from using UTF-16.

```
ISO-IR-165      [RFC1345]
VISCII
GB 12345
GB 18030 (**)   (see links bellow)
EUC-TW         (**)
```

are totally valid encodings but not registered at IANA. The names under which they are listed here are probably the most widely-known names for these encodings and are recommended names.

```
BIG5PLUS (**)

```

is a proprietary name.

Microsoft-related naming mess

Microsoft products misuse the following names:

KS_C_5601-1987

Microsoft extension to EUC-KR.

Proper names: CP949, UHC, x-windows-949 (as used by Mozilla).

See <http://lists.w3.org/Archives/Public/ietf-charsets/2001AprJun/0033.html> for details.

Encode aliases KS_C_5601-1987 to cp949 to reflect this common misuse. *Raw* KS_C_5601-1987 encoding is available as kcs5601-raw.

See [Encode::KR](#) for details.

GB2312

Microsoft extension to EUC-CN.

Proper names: CP936, GBK.

GB2312 has been registered in the EUC-CN meaning at IANA. This has partially repaired the situation: Microsoft's GB2312 has become a superset of the official GB2312.

Encode aliases GB2312 to euc-cn in full agreement with IANA registration. cp936 is supported separately. *Raw* GB_2312-80 encoding is available as gb2312-raw.

See [Encode::CN](#) for details.

Big5

Microsoft extension to Big5.

Proper name: CP950.

Encode separately supports Big5 and cp950.

Shift_JIS

Microsoft's understanding of Shift_JIS.

JIS has not endorsed the full Microsoft standard however. The official Shift_JIS includes only JIS X 0201 and JIS X 0208 character sets, while Microsoft has always used Shift_JIS to encode a wider character repertoire. See IANA registration for Windows-31J.

As a historical predecessor, Microsoft's variant probably has more rights for the name, though it may be objected that Microsoft shouldn't have used JIS as part of the name in the first place.

Unambiguous name: CP932. IANA name (not used?): Windows-31J.

Encode separately supports Shift_JIS and cp932.

Glossary

character repertoire

A collection of unique characters. A *character* set in the strictest sense. At this stage, characters are not numbered.

coded character set (CCS)

A character set that is mapped in a way computers can use directly. Many character encodings, including EUC, fall in this category.

character encoding scheme (CES)

An algorithm to map a character set to a byte sequence. You don't have to be able to tell which character set a given byte sequence belongs to. 7-bit ISO-2022 is a CES but it cannot be a CCS. EUC is an example of being both a CCS and CES.

charset (in MIME context)

has long been used in the meaning of encoding, CES.

While the word combination `character set` has lost this meaning in MIME context since [RFC 2130], the `charset` abbreviation has retained it. This is how [RFC 2277] and [RFC 2278] bless `charset`:

This document uses the term "charset" to mean a set of rules for mapping from a sequence of octets to a sequence of characters, such as the combination of a coded character set and a character encoding scheme; this is also what is used as an identifier in MIME "charset=" parameters, and registered in the IANA charset registry ... (Note that this is NOT a term used by other standards bodies, such as ISO).
[RFC 2277]

EUC

Extended Unix Character. See ISO-2022.

ISO-2022

A CES that was carefully designed to coexist with ASCII. There are a 7 bit version and an 8 bit version.

The 7 bit version switches character set via escape sequence so it cannot form a CCS. Since this is more difficult to handle in programs than the 8 bit version, the 7 bit version is not very popular except for iso-2022-jp, the *de facto* standard CES for e-mails.

The 8 bit version can form a CCS. EUC and ISO-8859 are two examples thereof. Pre-5.6 perl could use them as string literals.

UCS

Short for *Universal Character Set*. When you say just UCS, it means *Unicode*.

UCS-2

ISO/IEC 10646 encoding form: Universal Character Set coded in two octets.

Unicode

A character set that aims to include all character repertoires of the world. Many character sets in various national as well as industrial standards have become, in a way, just subsets of Unicode.

UTF

Short for *Unicode Transformation Format*. Determines how to map a Unicode character into a byte sequence.

UTF-16

A UTF in 16-bit encoding. Can either be in big endian or little endian. The big endian version is called UTF-16BE (equal to UCS-2 + surrogate support) and the little endian version is called UTF-16LE.

See Also

[Encode](#), [Encode::Byte](#), [Encode::CN](#), [Encode::JP](#), [Encode::KR](#), [Encode::TW](#), [Encode::EBCDIC](#), [Encode::Symbol](#), [Encode::MIME::Header](#), [Encode::Guess](#)

References

ECMA

European Computer Manufacturers Association <http://www.ecma.ch>

ECMA-035 (eq ISO-2022)

<http://www.ecma.ch/ecma1/STAND/ECMA-035.HTM>

The specification of ISO-2022 is available from the link above.

IANA

Internet Assigned Numbers Authority <http://www.iana.org/>

Assigned Charset Names by IANA

<http://www.iana.org/assignments/character-sets>

Most of the `canonical` names in `Encode` derive from this list so you can directly apply the string you have extracted from MIME header of mails and web pages.

ISO International Organization for Standardization <http://www.iso.ch/>

RFC

Request For Comments — need I say more? <http://www.rfc-editor.org/>, <http://www.rfc.net/>, <http://www.faqs.org/rfcs/>

UC Unicode Consortium <http://www.unicode.org/>

Unicode Glossary

<http://www.unicode.org/glossary/>

The glossary of this document is based upon this site.

Other Notable Sites

czyborra.com

<http://czyborra.com/>

Contains a a lot of useful information, especially gory details of ISO vs. vendor mappings.

CJK.inf

http://www.oreilly.com/people/authors/lunde/cjk_inf.html

Somewhat obsolete (last update in 1996), but still useful. Also try

ftp://ftp.oreilly.com/pub/examples/nutshell/cjk/pdf/GB18030_Summary.pdf

You will find brief info on EUC-CN, GBK and mostly on GB 18030.

Jungshik Shin's Hangul FAQ

<http://jshin.net/faq>

And especially its subject 8.

<http://jshin.net/faq/qa8.html>

A comprehensive overview of the Korean (KS *) standards.

debian.org: "Introduction to i18n"

A brief description for most of the mentioned CJK encodings is contained in

<http://www.debian.org/doc/manuals/intro-i18n/ch-codes.en.html>

Offline sources

CJKV Information Processing by Ken Lunde

CJKV Information Processing 1999 O'Reilly & Associates, ISBN : 1-56592-224-7

The modern successor of CJK.inf.

Features a comprehensive coverage of CJKV character sets and encodings along with many other issues faced by anyone trying to better support CJKV languages/scripts in all the areas of information processing.

To purchase this book, visit <http://www.oreilly.com/catalog/cjkvinfo/> or your favourite bookstore.

NAME

Encode::Symbol – Symbol Encodings

SYNOPSIS

```
use Encode qw/encode decode/;
$symbol = encode("symbol", $utf8); # loads Encode::Symbol implicitly
$utf8 = decode("", $symbol);      # ditto
```

ABSTRACT

This module implements symbol and dingbats encodings. Encodings supported are as follows.

Canonical	Alias	Description
-----------	-------	-------------

symbol		
dingbats		
AdobeZDingbat		
AdobeSymbol		
MacDingbats		

DESCRIPTION

To find out how to use this module in detail, see [Encode](#).

SEE ALSO

[Encode](#)

NAME

Encode::TW – Taiwan-based Chinese Encodings

SYNOPSIS

```
use Encode qw/encode decode/;
$big5 = encode("big5", $utf8); # loads Encode::TW implicitly
$utf8 = decode("big5", $big5); # ditto
```

DESCRIPTION

This module implements tradition Chinese charset encodings as used in Taiwan and Hong Kong. Encodings supported are as follows.

Canonical	Alias	Description

big5-eten	/\bbig-?5\$/i /\bbig5-?et(en)?\$/i /\btca-?big5\$/i	Big5 encoding (with ETen extensions)
big5-hkscs	/\bbig5-?hk(scs)?\$/i /\bhk(scs)?-?big5\$/i	Big5 + Cantonese characters in Hong Kong
MacChineseTrad		Big5 + Apple Vendor Mappings
cp950		Code Page 950 = Big5 + Microsoft vendor mappings

To find out how to use this module in detail, see [Encode](#).

NOTES

Due to size concerns, EUC-TW (Extended Unix Character), CCCII (Chinese Character Code for Information Interchange), BIG5PLUS (CMEX's Big5+) and BIG5EXT (CMEX's Big5e) are distributed separately on CPAN, under the name [Encode::HanExtra](#). That module also contains extra China-based encodings.

BUGS

Since the original big5 encoding (1984) is not supported anywhere (glibc and DOS-based systems uses big5 to mean big5-eten; Microsoft uses big5 to mean cp950), a conscious decision was made to alias big5 to big5-eten, which is the de facto superset of the original big5.

The CNS11643 encoding files are not complete. For common CNS11643 manipulation, please use EUC-TW in [Encode::HanExtra](#), which contains planes 1–7.

The ASCII region (0x00–0x7f) is preserved for all encodings, even though this conflicts with mappings by the Unicode Consortium. See

<http://www.debian.or.jp/~kubota/unicode-symbols.html.en>

to find out why it is implemented that way.

SEE ALSO

[Encode](#)

NAME

English – use nice English (or awk) names for ugly punctuation variables

SYNOPSIS

```
use English qw( -no_match_vars ) ; # Avoids regex performance penalty
use English;
...
if ( $ERRNO =~ /denied/ ) { ... }
```

DESCRIPTION

This module provides aliases for the built-in variables whose names no one seems to like to read. Variables with side-effects which get triggered just by accessing them (like `$0`) will still be affected.

For those variables that have an **awk** version, both long and short English alternatives are provided. For example, the `$/` variable can be referred to either `$RS` or `$INPUT_RECORD_SEPARATOR` if you are using the English module.

See [perlvar](#) for a complete list of these.

PERFORMANCE

This module can provoke sizeable inefficiencies for regular expressions, due to unfortunate implementation details. If performance matters in your application and you don't need `$PREMATCH`, `$MATCH`, or `$POSTMATCH`, try doing

```
use English qw( -no_match_vars ) ;
```

. **It is especially important to do this in modules to avoid penalizing all applications which use them.**

NAME

Env – perl module that imports environment variables as scalars or arrays

SYNOPSIS

```
use Env;
use Env qw(PATH HOME TERM);
use Env qw($SHELL @LD_LIBRARY_PATH);
```

DESCRIPTION

Perl maintains environment variables in a special hash named %ENV. For when this access method is inconvenient, the Perl module Env allows environment variables to be treated as scalar or array variables.

The `Env::import()` function ties environment variables with suitable names to global Perl variables with the same names. By default it ties all existing environment variables (`keys %ENV`) to scalars. If the `import` function receives arguments, it takes them to be a list of variables to tie; it's okay if they don't yet exist. The scalar type prefix '\$' is inferred for any element of this list not prefixed by '\$' or '@'. Arrays are implemented in terms of `split` and `join`, using `$Config::Config{path_sep}` as the delimiter.

After an environment variable is tied, merely use it like a normal variable. You may access its value

```
@path = split(/:/, $PATH);
print join("\n", @LD_LIBRARY_PATH), "\n";
```

or modify it

```
$PATH .= " . ";
push @LD_LIBRARY_PATH, $dir;
```

however you'd like. Bear in mind, however, that each access to a tied array variable requires splitting the environment variable's string anew.

The code:

```
use Env qw(@PATH);
push @PATH, ' . ';
```

is equivalent to:

```
use Env qw(PATH);
$PATH .= " . ";
```

except that if `$ENV{PATH}` started out empty, the second approach leaves it with the (odd) value `" . "`, but the first approach leaves it with `" . "`.

To remove a tied environment variable from the environment, assign it the undefined value

```
undef $PATH;
undef @LD_LIBRARY_PATH;
```

LIMITATIONS

On VMS systems, arrays tied to environment variables are read-only. Attempting to change anything will cause a warning.

AUTHOR

Chip Salzenberg <chip@fin.uucp> and Gregor N. Purdy <gregor@focusresearch.com>

NAME

Exporter – Implements default import method for modules

SYNOPSIS

In module YourModule.pm:

```
package YourModule;
require Exporter;
@ISA = qw(Exporter);
@EXPORT_OK = qw(munge frobnicate); # symbols to export on request
```

In other files which wish to use YourModule:

```
use ModuleName qw(frobnicate);      # import listed symbols
frobnicate ($left, $right)          # calls YourModule::frobnicate
```

DESCRIPTION

The Exporter module implements an import method which allows a module to export functions and variables to its users' namespaces. Many modules use Exporter rather than implementing their own import method because Exporter provides a highly flexible interface, with an implementation optimised for the common case.

Perl automatically calls the import method when processing a use statement for a module. Modules and use are documented in [perlfunc](#) and [perlmod](#). Understanding the concept of modules and how the use statement operates is important to understanding the Exporter.

How to Export

The arrays @EXPORT and @EXPORT_OK in a module hold lists of symbols that are going to be exported into the users name space by default, or which they can request to be exported, respectively. The symbols can represent functions, scalars, arrays, hashes, or typeglobs. The symbols must be given by full name with the exception that the ampersand in front of a function is optional, e.g.

```
@EXPORT      = qw(afunc $scalar @array); # afunc is a function
@EXPORT_OK   = qw(&bfunc %hash *typeglob); # explicit prefix on &bfunc
```

If you are only exporting function names it is recommended to omit the ampersand, as the implementation is faster this way.

Selecting What To Export

Do **not** export method names!

Do **not** export anything else by default without a good reason!

Exports pollute the namespace of the module user. If you must export try to use @EXPORT_OK in preference to @EXPORT and avoid short or common symbol names to reduce the risk of name clashes.

Generally anything not exported is still accessible from outside the module using the ModuleName::item_name (or \$blessed_ref->method) syntax. By convention you can use a leading underscore on names to informally indicate that they are 'internal' and not for public use.

(It is actually possible to get private functions by saying:

```
my $subref = sub { ... };
$subref->(@args);          # Call it as a function
$obj->$subref(@args);      # Use it as a method
```

However if you use them for methods it is up to you to figure out how to make inheritance work.)

As a general rule, if the module is trying to be object oriented then export nothing. If it's just a collection of functions then @EXPORT_OK anything but use @EXPORT with caution. For function and method names use barewords in preference to names prefixed with ampersands for the export lists.

Other module design guidelines can be found in [perlmod](#).

How to Import

In other files which wish to use your module there are three basic ways for them to load your module and import its symbols:

```
use ModuleName;
```

This imports all the symbols from ModuleName's @EXPORT into the namespace of the use statement.

```
use ModuleName ();
```

This causes perl to load your module but does not import any symbols.

```
use ModuleName qw(...);
```

This imports only the symbols listed by the caller into their namespace. All listed symbols must be in your @EXPORT or @EXPORT_OK, else an error occurs. The advanced export features of Exporter are accessed like this, but with list entries that are syntactically distinct from symbol names.

Unless you want to use its advanced features, this is probably all you need to know to use Exporter.

Advanced features

Specialised Import Lists

If the first entry in an import list begins with !, : or / then the list is treated as a series of specifications which either add to or delete from the list of names to import. They are processed left to right. Specifications are in the form:

[!]name	This name only
[!]:DEFAULT	All names in @EXPORT
[!]:tag	All names in \$EXPORT_TAGS{tag} anonymous list
[!]/pattern/	All names in @EXPORT and @EXPORT_OK which match

A leading ! indicates that matching names should be deleted from the list of names to import. If the first specification is a deletion it is treated as though preceded by :DEFAULT. If you just want to import extra names in addition to the default set you will still need to include :DEFAULT explicitly.

e.g., Module.pm defines:

```
@EXPORT      = qw(A1 A2 A3 A4 A5);
@EXPORT_OK   = qw(B1 B2 B3 B4 B5);
%EXPORT_TAGS = (T1 => [qw(A1 A2 B1 B2)], T2 => [qw(A1 A2 B3 B4)]);
```

Note that you cannot use tags in @EXPORT or @EXPORT_OK.

Names in EXPORT_TAGS must also appear in @EXPORT or @EXPORT_OK.

An application using Module can say something like:

```
use Module qw(:DEFAULT :T2 !B3 A3);
```

Other examples include:

```
use Socket qw(!/^ [AP] F_ / !SOMAXCONN !SOL_SOCKET);
use POSIX  qw(:errno_h :termios_h !TCSADRAIN !/^EXIT/);
```

Remember that most patterns (using //) will need to be anchored with a leading ^, e.g., /^EXIT/ rather than /EXIT/.

You can say BEGIN { \$Exporter::Verbose=1 } to see how the specifications are being processed and what is actually being imported into modules.

Exporting without using Exporter's import method

Exporter has a special method, 'export_to_level' which is used in situations where you can't directly call Exporter's import method. The export_to_level method looks like:

```
MyPackage->export_to_level($where_to_export, $package, @what_to_export);
```

where \$where_to_export is an integer telling how far up the calling stack to export your symbols, and @what_to_export is an array telling what symbols *to* export (usually this is @_). The \$package argument is currently unused.

For example, suppose that you have a module, A, which already has an import function:

```
package A;

@ISA = qw(Exporter);
@EXPORT_OK = qw ($b);

sub import
{
    $A::b = 1;      # not a very useful import method
}
```

and you want to Export symbol \$A::b back to the module that called package A. Since Exporter relies on the import method to work, via inheritance, as it stands Exporter::import() will never get called. Instead, say the following:

```
package A;
@ISA = qw(Exporter);
@EXPORT_OK = qw ($b);

sub import
{
    $A::b = 1;
    A->export_to_level(1, @_);
}
```

This will export the symbols one level 'above' the current package – ie: to the program or module that used package A.

Note: Be careful not to modify '@_' at all before you call export_to_level – or people using your package will get very unexplained results!

Module Version Checking

The Exporter module will convert an attempt to import a number from a module into a call to \$module_name->require_version(\$value). This can be used to validate that the version of the module being used is greater than or equal to the required version.

The Exporter module supplies a default require_version method which checks the value of \$VERSION in the exporting module.

Since the default require_version method treats the \$VERSION number as a simple numeric value it will regard version 1.10 as lower than 1.9. For this reason it is strongly recommended that you use numbers with at least two decimal places, e.g., 1.09.

Managing Unknown Symbols

In some situations you may want to prevent certain symbols from being exported. Typically this applies to extensions which have functions or constants that may not exist on some systems.

The names of any symbols that cannot be exported should be listed in the @EXPORT_FAIL array.

If a module attempts to import any of these symbols the Exporter will give the module an opportunity to handle the situation before generating an error. The Exporter will call an export_fail method with a list of the

failed symbols:

```
@failed_symbols = $module_name->export_fail(@failed_symbols);
```

If the `export_fail` method returns an empty list then no error is recorded and all the requested symbols are exported. If the returned list is not empty then an error is generated for each symbol and the export fails. The Exporter provides a default `export_fail` method which simply returns the list unchanged.

Uses for the `export_fail` method include giving better error messages for some symbols and performing lazy architectural checks (put more symbols into `@EXPORT_FAIL` by default and then take them out if someone actually tries to use them and an expensive check shows that they are usable on that platform).

Tag Handling Utility Functions

Since the symbols listed within `%EXPORT_TAGS` must also appear in either `@EXPORT` or `@EXPORT_OK`, two utility functions are provided which allow you to easily add tagged sets of symbols to `@EXPORT` or `@EXPORT_OK`:

```
%EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

Exporter::export_tags('foo');      # add aa, bb and cc to @EXPORT
Exporter::export_ok_tags('bar');    # add aa, cc and dd to @EXPORT_OK
```

Any names which are not tags are added to `@EXPORT` or `@EXPORT_OK` unchanged but will trigger a warning (with `-w`) to avoid misspelt tags names being silently added to `@EXPORT` or `@EXPORT_OK`. Future versions may make this a fatal error.

Generating combined tags

If several symbol categories exist in `%EXPORT_TAGS`, it's usually useful to create the utility `":all"` to simplify `"use"` statements.

The simplest way to do this is:

```
%EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

# add all the other ":class" tags to the ":all" class,
# deleting duplicates
{
    my %seen;

    push @{$EXPORT_TAGS{all}},
        grep {!$seen{$_}++} @{$EXPORT_TAGS{$_}} foreach keys %EXPORT_TAGS;
}
```

`CGI.pm` creates an `":all"` tag which contains some (but not really all) of its categories. That could be done with one small change:

```
# add some of the other ":class" tags to the ":all" class,
# deleting duplicates
{
    my %seen;

    push @{$EXPORT_TAGS{all}},
        grep {!$seen{$_}++} @{$EXPORT_TAGS{$_}}
            foreach qw/html2 html3 netscape form cgi internal/;
}
```

Note that the tag names in `%EXPORT_TAGS` don't have the leading `':'`.

AUTOLOADED Constants

Many modules make use of `AUTOLOADING` for constant subroutines to avoid having to compile and waste memory on rarely used values (see [perlsyn](#) for details on constant subroutines). Calls to such constant subroutines are not optimized away at compile time because they can't be checked at compile time for

constancy.

Even if a prototype is available at compile time, the body of the subroutine is not (it hasn't been AUTOLOADED yet). perl needs to examine both the () prototype and the body of a subroutine at compile time to detect that it can safely replace calls to that subroutine with the constant value.

A workaround for this is to call the constants once in a BEGIN block:

```
package My ;  
use Socket ;  
  
foo( SO_LINGER );      ## SO_LINGER NOT optimized away; called at runtime  
BEGIN { SO_LINGER }  
foo( SO_LINGER );      ## SO_LINGER optimized away at compile time.
```

This forces the AUTOLOAD for SO_LINGER to take place before SO_LINGER is encountered later in My package.

If you are writing a package that AUTOLOADs, consider forcing an AUTOLOAD for any constants explicitly imported by other packages or which are usually used when your package is used.

NAME

Exporter::Heavy – Exporter guts

SYNOPSIS

(internal use only)

DESCRIPTION

No user-serviceable parts inside.

NAME

ExtUtils::Command – utilities to replace common UNIX commands in Makefiles etc.

SYNOPSIS

```
perl -MExtUtils::Command -e cat files... > destination
perl -MExtUtils::Command -e mv source... destination
perl -MExtUtils::Command -e cp source... destination
perl -MExtUtils::Command -e touch files...
perl -MExtUtils::Command -e rm_f file...
perl -MExtUtils::Command -e rm_rf directories...
perl -MExtUtils::Command -e mkpath directories...
perl -MExtUtils::Command -e eqtime source destination
perl -MExtUtils::Command -e chmod mode files...
perl -MExtUtils::Command -e test_f file
```

DESCRIPTION

The module is used to replace common UNIX commands. In all cases the functions work from @ARGV rather than taking arguments. This makes them easier to deal with in Makefiles.

```
perl -MExtUtils::Command -e some_command some files to work on
```

NOT

```
perl -MExtUtils::Command -e 'some_command qw(some files to work on)'
```

Filenames with * and ? will be glob expanded.

cat Concatenates all files mentioned on command line to STDOUT.

eqtime src dst

Sets modified time of dst to that of src

rm_rf files....

Removes directories – recursively (even if readonly)

rm_f files....

Removes files (even if readonly)

touch files ...

Makes files exist, with current timestamp

mv source... destination

Moves source to destination. Multiple sources are allowed if destination is an existing directory.

cp source... destination

Copies source to destination. Multiple sources are allowed if destination is an existing directory.

chmod mode files...

Sets UNIX like permissions ‘mode’ on all the files.

mkpath directory...

Creates directory, including any parent directories.

test_f file

Tests if a file exists

BUGS

Should probably be Auto/Self loaded.

SEE ALSO

ExtUtils::MakeMaker, ExtUtils::MM_Unix, ExtUtils::MM_Win32

AUTHOR

Nick Ing-Simmons <*nick@ni-s.u-net.com*>.

NAME

ExtUtils::Command::MM – Commands for the MM's to use in Makefiles

SYNOPSIS

```
perl -MExtUtils::Command::MM -e "function" files...
```

DESCRIPTION

FOR INTERNAL USE ONLY! The interface is not stable.

ExtUtils::Command::MM encapsulates code which would otherwise have to be done with large "one" liners.

They all read their input from @ARGV unless otherwise noted.

Any \$(FOO) used in the examples are make variables, not Perl.

test_harness

```
test_harness($verbose, @test_libs);
```

Runs the tests on @ARGV via Test::Harness passing through the \$verbose flag. Any @test_libs will be unshifted onto the test's @INC.

@test_libs are run in alphabetical order.

NAME

ExtUtils::Constant – generate XS code to import C header constants

SYNOPSIS

```
use ExtUtils::Constant qw (WriteConstants);
WriteConstants(
    NAME => 'Foo',
    NAMES => [qw(FOO BAR BAZ)],
);
# Generates wrapper code to make the values of the constants FOO BAR BAZ
# available to perl
```

DESCRIPTION

ExtUtils::Constant facilitates generating C and XS wrapper code to allow perl modules to AUTOLOAD constants defined in C library header files. It is principally used by the h2xs utility, on which this code is based. It doesn't contain the routines to scan header files to extract these constants.

USAGE

Generally one only needs to call the WriteConstants function, and then

```
#include "const-c.inc"
```

in the C section of Foo.xs

```
INCLUDE const-xs.inc
```

in the XS section of Foo.xs.

For greater flexibility use constant_types(), C_constant and XS_constant, with which WriteConstants is implemented.

Currently this module understands the following types. h2xs may only know a subset. The sizes of the numeric types are chosen by the Configure script at compile time.

IV signed integer, at least 32 bits.

UV unsigned integer, the same size as IV

NV floating point type, probably double, possibly long double

PV NUL terminated string, length will be determined with strlen

PVN

A fixed length thing, given as a [pointer, length] pair. If you know the length of a string at compile time you may use this instead of PV

SV A mortal SV.

YES

Truth. (PL_sv_yes) The value is not needed (and ignored).

NO Defined Falsehood. (PL_sv_no) The value is not needed (and ignored).

UNDEF

undef. The value of the macro is not needed.

FUNCTIONS

C_stringify NAME

A function which returns a 7 bit ASCII correctly \ escaped version of the string passed suitable for C's "" or ". It will die if passed Unicode characters.

perl_stringify NAME

A function which returns a 7 bit ASCII correctly \ escaped version of the string passed suitable for a perl "" string.

constant_types

A function returning a single scalar with #define definitions for the constants used internally between the generated C and XS functions.

memEQ_clause NAME, CHECKED_AT, INDENT

A function to return a suitable C if statement to check whether *NAME* is equal to the C variable name. If *CHECKED_AT* is defined, then it is used to avoid memEQ for short names, or to generate a comment to highlight the position of the character in the switch statement.

assign INDENT, TYPE, PRE, POST, VALUE...

A function to return a suitable assignment clause. If *TYPE* is aggregate (eg *PVN* expects both pointer and length) then there should be multiple *VALUES* for the components. *PRE* and *POST* if defined give snippets of C code to proceed and follow the assignment. *PRE* will be at the start of a block, so variables may be defined in it.

return_clause

return_clause ITEM, INDENT

A function to return a suitable #ifdef clause. *ITEM* is a hashref (as passed to C_constant and match_clause. *INDENT* is the number of spaces to indent, defaulting to 6.

XXX document me

switch_clause INDENT, NAMELEN, ITEMHASH, ITEM...

An internal function to generate a suitable switch clause, called by C_constant *ITEMs* are in the hash ref format as given in the description of C_constant, and must all have the names of the same length, given by *NAMELEN* (This is not checked). *ITEMHASH* is a reference to a hash, keyed by name, values being the hashrefs in the *ITEM* list. (No parameters are modified, and there can be keys in the *ITEMHASH* that are not in the list of *ITEMs* without causing problems).

params WHAT

An internal function. *WHAT* should be a hashref of types the constant function will return. *params* returns a hashref keyed IV NV PV SV to show which combination of pointers will be needed in the C argument list.

dump_names

dump_names DEFAULT_TYPE, TYPES, INDENT, OPTIONS, ITEM...

An internal function to generate the embedded perl code that will regenerate the constant subroutines. *DEFAULT_TYPE*, *TYPES* and *ITEMs* are the same as for C_constant. *INDENT* is treated as number of spaces to indent by. *OPTIONS* is a hashref of options. Currently only declare_types is recognised. If the value is true a \$types is always declared in the perl code generated, if defined and false never declared, and if undefined \$types is only declared if the values in *TYPES* as passed in cannot be inferred from *DEFAULT_TYPES* and the *ITEMs*.

dogfood

dogfood PACKAGE, SUBNAME, DEFAULT_TYPE, TYPES, INDENT, BREAKOUT, ITEM...

An internal function to generate the embedded perl code that will regenerate the constant subroutines. Parameters are the same as for C_constant.

C_constant

C_constant PACKAGE, SUBNAME, DEFAULT_TYPE, TYPES, INDENT, BREAKOUT, ITEM...

A function that returns a **list** of C subroutine definitions that return the value and type of constants

when passed the name by the XS wrapper. *ITEM...* gives a list of constant names. Each can either be a string, which is taken as a C macro name, or a reference to a hash with the following keys

name	The name of the constant, as seen by the perl code.
type	The type of the constant (<i>IV</i> , <i>NV</i> etc)
value	A C expression for the value of the constant, or a list of C expressions if the type is aggregate. This defaults to the <i>name</i> if not given.
macro	The C pre-processor macro to use in the <code>#ifdef</code> . This defaults to the <i>name</i> , and is mainly used if <i>value</i> is an enum. If a reference an array is passed then the first element is used in place of the <code>#ifdef</code> line, and the second element in place of the <code>#endif</code> . This allows pre-processor constructions such as

```
#if defined (foo)
#if !defined (bar)
...
#endif
#endif
```

to be used to determine if a constant is to be defined.

A "macro" 1 signals that the constant is always defined, so the `#if/#endif` test is omitted.

default	Default value to use (instead of croaking with "your vendor has not defined...") to return if the macro isn't defined. Specify a reference to an array with type followed by value(s).
pre	C code to use before the assignment of the value of the constant. This allows you to use temporary variables to extract a value from part of a <code>struct</code> and return this as <i>value</i> . This C code is places at the start of a block, so you can declare variables in it.
post	C code to place between the assignment of value (to a temporary) and the return from the function. This allows you to clear up anything in <i>pre</i> . Rarely needed.
def_pre	
def_post	Equivalents of <i>pre</i> and <i>post</i> for the default value.

utf8 Generated internally. Is zero or undefined if name is 7 bit ASCII, "no" if the name is 8 bit (and so should only match if `SvUTF8()` is false), "yes" if the name is utf8 encoded.

The internals automatically clone any name with characters 128–255 but none 256+ (ie one that could be either in bytes or utf8) into a second entry which is utf8 encoded.

PACKAGE is the name of the package, and is only used in comments inside the generated C code.

The next 5 arguments can safely be given as `undef`, and are mainly used for recursion. *SUBNAME* defaults to `constant` if undefined.

DEFAULT_TYPE is the type returned by *ITEMs* that don't specify their type. In turn it defaults to *IV*. *TYPES* should be given either as a comma separated list of types that the C subroutine `constant` will generate or as a reference to a hash. *DEFAULT_TYPE* will be added to the list if not present, as will any types given in the list of *ITEMs*. The resultant list should be the same list of types that `XS_constant` is given. [Otherwise `XS_constant` and `C_constant` may differ in the number of parameters to the constant function. *INDENT* is currently unused and ignored. In future it may be used to pass in information used to change the C indentation style used.] The best way to maintain consistency is to pass in a hash reference and let this function update it.

BREAKOUT governs when child functions of *SUBNAME* are generated. If there are *BREAKOUT* or more *ITEMs* with the same length of name, then the code to switch between them is placed into a function named *SUBNAME_LEN*, for example `constant_5` for names 5 characters long. The default *BREAKOUT* is 3. A single *ITEM* is always inlined.

XS_constant PACKAGE, TYPES, SUBNAME, C_SUBNAME

A function to generate the XS code to implement the perl subroutine *PACKAGE::constant* used by *PACKAGE::AUTOLOAD* to load constants. This XS code is a wrapper around a C subroutine usually generated by *C_constant*, and usually named *constant*.

TYPES should be given either as a comma separated list of types that the C subroutine *constant* will generate or as a reference to a hash. It should be the same list of types as *C_constant* was given. [Otherwise *XS_constant* and *C_constant* may have different ideas about the number of parameters passed to the C function *constant*]

You can call the perl visible subroutine something other than *constant* if you give the parameter *SUBNAME*. The C subroutine it calls defaults to the name of the perl visible subroutine, unless you give the parameter *C_SUBNAME*.

autoload PACKAGE, VERSION, AUTOLOADER

A function to generate the AUTOLOAD subroutine for the module *PACKAGE VERSION* is the perl version the code should be backwards compatible with. It defaults to the version of perl running the subroutine. If *AUTOLOADER* is true, the AUTOLOAD subroutine falls back on *AutoLoader::AUTOLOAD* for all names that the *constant()* routine doesn't recognise.

WriteMakefileSnippet

WriteMakefileSnippet ATTRIBUTE => VALUE [, ...]

A function to generate perl code for *Makefile.PL* that will regenerate the constant subroutines. Parameters are named as passed to *WriteConstants*, with the addition of *INDENT* to specify the number of leading spaces (default 2).

Currently only *INDENT*, *NAME*, *DEFAULT_TYPE*, *NAMES*, *C_FILE* and *XS_FILE* are recognised.

WriteConstants ATTRIBUTE => VALUE [, ...]

Writes a file of C code and a file of XS code which you should *#include* and *INCLUDE* in the C and XS sections respectively of your module's XS code. You probably want to do this in your *Makefile.PL*, so that you can easily edit the list of constants without touching the rest of your module. The attributes supported are

NAME

Name of the module. This must be specified

DEFAULT_TYPE

The default type for the constants. If not specified IV is assumed.

BREAKOUT_AT

The names of the constants are grouped by length. Generate child subroutines for each group with this number or more names in.

NAMES

An array of constants' names, either scalars containing names, or hashrefs as detailed in *"C_constant"*.

C_FILE

The name of the file to write containing the C code. The default is *const-c.inc*. The *-* in the name ensures that the file can't be mistaken for anything related to a legitimate perl package name, and not naming the file *.c* avoids having to override *Makefile.PL*'s *.xs* to *.c* rules.

XS_FILE

The name of the file to write containing the XS code. The default is *const-xs.inc*.

SUBNAME

The perl visible name of the XS subroutine generated which will return the constants. The default is `constant`.

C_SUBNAME

The name of the C subroutine generated which will return the constants. The default is *SUBNAME*. Child subroutines have `_` and the name length appended, so constants with 10 character names would be in `constant_10` with the default *XS_SUBNAME*.

AUTHOR

Nicholas Clark <nick@ccl4.org> based on the code in `h2xs` by Larry Wall and others

NAME

ExtUtils::Embed – Utilities for embedding Perl in C/C++ applications

SYNOPSIS

```
perl -MExtUtils::Embed -e xsinit
perl -MExtUtils::Embed -e ccopts
perl -MExtUtils::Embed -e ldopts
```

DESCRIPTION

ExtUtils::Embed provides utility functions for embedding a Perl interpreter and extensions in your C/C++ applications. Typically, an application **Makefile** will invoke ExtUtils::Embed functions while building your application.

@EXPORT

ExtUtils::Embed exports the following functions:

```
xsinit(), ldopts(), ccopts(), perl_inc(), ccflags(), ccdlflags(), xsi_header(),
xsi_protos(), xsi_body()
```

FUNCTIONS

`xsinit()`

Generate C/C++ code for the XS initializer function.

When invoked as ``perl -MExtUtils::Embed -e xsinit -`` the following options are recognized:

`-o <output filename>` (Defaults to **perlxsi.c**)

`-o STDOUT` will print to STDOUT.

`-std` (Write code for extensions that are linked with the current Perl.)

Any additional arguments are expected to be names of modules to generate code for.

When invoked with parameters the following are accepted and optional:

```
xsinit($filename, $std, [@modules])
```

Where,

\$filename is equivalent to the `-o` option.

\$std is boolean, equivalent to the `-std` option.

[@modules] is an array ref, same as additional arguments mentioned above.

Examples

```
perl -MExtUtils::Embed -e xsinit -- -o xsinit.c Socket
```

This will generate code with an **xs_init** function that glues the perl **Socket::bootstrap** function to the C **boot_Socket** function and writes it to a file named **xsinit.c**.

Note that **DynaLoader** is a special case where it must call **boot_DynaLoader** directly.

```
perl -MExtUtils::Embed -e xsinit
```

This will generate code for linking with **DynaLoader** and each static extension found in **\$Config{static_ext}**. The code is written to the default file name **perlxsi.c**.

```
perl -MExtUtils::Embed -e xsinit -- -o xsinit.c -std DBI DBD::Oracle
```

Here, code is written for all the currently linked extensions along with code for **DBI** and **DBD::Oracle**.

If you have a working **DynaLoader** then there is rarely any need to statically link in any other extensions.

`ldopts()`

Output arguments for linking the Perl library and extensions to your application.

When invoked as `'perl -MExtUtils::Embed -e ldopts -'` the following options are recognized:

-std

Output arguments for linking the Perl library and any extensions linked with the current Perl.

-I <path1:path2>

Search path for ModuleName.a archives. Default path is **@INC**. Library archives are expected to be found as `/some/path/auto/ModuleName/ModuleName.a`. For example, when looking for **Socket.a** relative to a search path, we should find `auto/Socket/Socket.a`.

When looking for **DBD::Oracle** relative to a search path, we should find `auto/DBD/Oracle/Oracle.a`.

Keep in mind that you can always supply `/my/own/path/ModuleName.a` as an additional linker argument.

— <list of linker args>

Additional linker arguments to be considered.

Any additional arguments found before the — token are expected to be names of modules to generate code for.

When invoked with parameters the following are accepted and optional:

`ldopts($std, [@modules], [@link_args], $path)`

Where:

\$std is boolean, equivalent to the **-std** option.

[@modules] is equivalent to additional arguments found before the — token.

[@link_args] is equivalent to arguments found after the — token.

\$path is equivalent to the **-I** option.

In addition, when `ldopts` is called with parameters, it will return the argument string rather than print it to STDOUT.

Examples

```
perl -MExtUtils::Embed -e ldopts
```

This will print arguments for linking with **libperl.a**, **DynaLoader** and extensions found in `$Config{static_ext}`. This includes libraries found in `$Config{libs}` and the first ModuleName.a library for each extension that is found by searching **@INC** or the path specified by the **-I** option. In addition, when ModuleName.a is found, additional linker arguments are picked up from the `extralibs.ld` file in the same directory.

```
perl -MExtUtils::Embed -e ldopts -- -std Socket
```

This will do the same as the above example, along with printing additional arguments for linking with the **Socket** extension.

```
perl -MExtUtils::Embed -e ldopts -- DynaLoader
```

This will print arguments for linking with just the **DynaLoader** extension and **libperl.a**.

```
perl -MExtUtils::Embed -e ldopts -- -std Mysql -- -L/usr/mysql/lib -lmysql
```

Any arguments after the second ‘—’ token are additional linker arguments that will be examined for potential conflict. If there is no conflict, the additional arguments will be part of the output.

`perl_inc()`

For including perl header files this function simply prints:

```
-I$Config{archlibexp}/CORE
```

So, rather than having to say:

```
perl -MConfig -e 'print "-I$Config{archlibexp}/CORE"'
```

Just say:

```
perl -MExtUtils::Embed -e perl_inc
```

`ccflags()`, `ccdlflags()`

These functions simply print `$Config{ccflags}` and `$Config{ccdlflags}`

`ccopts()`

This function combines `perl_inc()`, `ccflags()` and `ccdlflags()` into one.

`xsi_header()`

This function simply returns a string defining the same **EXTERN_C** macro as **perlmain.c** along with #including **perl.h** and **EXTERN.h**.

`xsi_protos(@modules)`

This function returns a string of **boot_**`$ModuleName` prototypes for each @modules.

`xsi_body(@modules)`

This function returns a string of calls to **newXS()** that glue the module **bootstrap** function to **boot_**`ModuleName` for each @modules.

xsinit() uses the `xsi_*` functions to generate most of its code.

EXAMPLES

For examples on how to use **ExtUtils::Embed** for building C/C++ applications with embedded perl, see [perlembed](#).

SEE ALSO

[perlembed](#)

AUTHOR

Doug MacEachern <doug@osf.org>

Based on ideas from Tim Bunce <Tim.Bunce@ig.co.uk> and **minimod.pl** by Andreas Koenig <k@anna.in-berlin.de> and Tim Bunce.

NAME

ExtUtils::Install – install files from here to there

SYNOPSIS

```
use ExtUtils::Install;

install($hashref, $verbose, $nonono) ;

uninstall($packlistfile, $verbose, $nonono) ;

pm_to_blib($hashref) ;
```

DESCRIPTION

Both `install()` and `uninstall()` are specific to the way ExtUtils::MakeMaker handles the installation and deinstallation of perl modules. They are not designed as general purpose tools.

`install()` takes three arguments. A reference to a hash, a verbose switch and a don't-really-do-it switch. The hash ref contains a mapping of directories: each key/value pair is a combination of directories to be copied. Key is a directory to copy from, value is a directory to copy to. The whole tree below the "from" directory will be copied preserving timestamps and permissions.

There are two keys with a special meaning in the hash: "read" and "write". After the copying is done, `install` will write the list of target files to the file named by `$hashref->{write}`. If there is another file named by `$hashref->{read}`, the contents of this file will be merged into the written file. The read and the written file may be identical, but on AFS it is quite likely that people are installing to a different directory than the one where the files later appear.

`install_default()` takes one or less arguments. If no arguments are specified, it takes `$ARGV[0]` as if it was specified as an argument. The argument is the value of MakeMaker's FULLEXT key, like *Tk/Canvas*. This function calls `install()` with the same arguments as the defaults the MakeMaker would use.

The argument-less form is convenient for install scripts like

```
perl -MExtUtils::Install -e install_default Tk/Canvas
```

Assuming this command is executed in a directory with a populated *blib* directory, it will proceed as if the *blib* was build by MakeMaker on this machine. This is useful for binary distributions.

`uninstall()` takes as first argument a file containing filenames to be unlinked. The second argument is a verbose switch, the third is a no-don't-really-do-it-now switch.

`pm_to_blib()` takes a hashref as the first argument and copies all keys of the hash to the corresponding values efficiently. Filenames with the extension pm are autosplit. Second argument is the autosplit directory. If third argument is not empty, it is taken as a filter command to be ran on each .pm file, the output of the command being what is finally copied, and the source for auto-splitting.

You can have an environment variable PERL_INSTALL_ROOT set which will be prepended as a directory to each installed file (and directory).

NAME

ExtUtils::Installed – Inventory management of installed modules

SYNOPSIS

```
use ExtUtils::Installed;
my ($inst) = ExtUtils::Installed->new();
my (@modules) = $inst->modules();
my (@missing) = $inst->validate("DBI");
my $all_files = $inst->files("DBI");
my $files_below_usr_local = $inst->files("DBI", "all", "/usr/local");
my $all_dirs = $inst->directories("DBI");
my $dirs_below_usr_local = $inst->directory_tree("DBI", "prog");
my $packlist = $inst->packlist("DBI");
```

DESCRIPTION

ExtUtils::Installed provides a standard way to find out what core and module files have been installed. It uses the information stored in .packlist files created during installation to provide this information. In addition it provides facilities to classify the installed files and to extract directory information from the .packlist files.

USAGE

The new() function searches for all the installed .packlists on the system, and stores their contents. The .packlists can be queried with the functions described below.

FUNCTIONS

new()

This takes no parameters, and searches for all the installed .packlists on the system. The packlists are read using the ExtUtils::packlist module.

modules()

This returns a list of the names of all the installed modules. The perl 'core' is given the special name 'Perl'.

files()

This takes one mandatory parameter, the name of a module. It returns a list of all the filenames from the package. To obtain a list of core perl files, use the module name 'Perl'. Additional parameters are allowed. The first is one of the strings "prog", "doc" or "all", to select either just program files, just manual files or all files. The remaining parameters are a list of directories. The filenames returned will be restricted to those under the specified directories.

directories()

This takes one mandatory parameter, the name of a module. It returns a list of all the directories from the package. Additional parameters are allowed. The first is one of the strings "prog", "doc" or "all", to select either just program directories, just manual directories or all directories. The remaining parameters are a list of directories. The directories returned will be restricted to those under the specified directories. This method returns only the leaf directories that contain files from the specified module.

directory_tree()

This is identical in operation to directories(), except that it includes all the intermediate directories back up to the specified directories.

validate()

This takes one mandatory parameter, the name of a module. It checks that all the files listed in the modules .packlist actually exist, and returns a list of any missing files. If an optional second argument which evaluates to true is given any missing files will be removed from the .packlist

packlist()

This returns the ExtUtils::Packlist object for the specified module.

version()

This returns the version number for the specified module.

EXAMPLE

See the example in [ExtUtils::Packlist](#).

AUTHOR

Alan Burlison <Alan.Burlison@uk.sun.com>

NAME

ExtUtils::Liblist – determine libraries to use and how to use them

SYNOPSIS

```
require ExtUtils::Liblist;

ExtUtils::Liblist::ext($self, $potential_libs, $verbose, $need_names);
```

DESCRIPTION

This utility takes a list of libraries in the form `-llib1 -llib2 -llib3` and returns lines suitable for inclusion in an extension Makefile. Extra library paths may be included with the form `-L/another/path` this will affect the searches for all subsequent libraries.

It returns an array of four or five scalar values: `EXTRALIBS`, `BSLOADLIBS`, `LDLOADLIBS`, `LD_RUN_PATH`, and, optionally, a reference to the array of the filenames of actual libraries. Some of these don't mean anything unless on Unix. See the details about those platform specifics below. The list of the filenames is returned only if `$need_names` argument is true.

Dependent libraries can be linked in one of three ways:

- For static extensions
by the `ld` command when the perl binary is linked with the extension library. See `EXTRALIBS` below.
- For dynamic extensions at build/link time
by the `ld` command when the shared object is built/linked. See `LDLOADLIBS` below.
- For dynamic extensions at load time
by the DynaLoader when the shared object is loaded. See `BSLOADLIBS` below.

EXTRALIBS

List of libraries that need to be linked with when linking a perl binary which includes this extension. Only those libraries that actually exist are included. These are written to a file and used when linking perl.

LDLOADLIBS and LD_RUN_PATH

List of those libraries which can or must be linked into the shared library when created using `ld`. These may be static or dynamic libraries. `LD_RUN_PATH` is a colon separated list of the directories in `LDLOADLIBS`. It is passed as an environment variable to the process that links the shared library.

BSLOADLIBS

List of those libraries that are needed but can be linked in dynamically at run time on this platform. SunOS/Solaris does not need this because `ld` records the information (from `LDLOADLIBS`) into the object file. This list is used to create a `.bs` (bootstrap) file.

PORTABILITY

This module deals with a lot of system dependencies and has quite a few architecture specific `ifs` in the code.

VMS implementation

The version of `ext()` which is executed under VMS differs from the Unix-OS/2 version in several respects:

- Input library and path specifications are accepted with or without the `-l` and `-L` prefixes used by Unix linkers. If neither prefix is present, a token is considered a directory to search if it is in fact a directory, and a library to search for otherwise. Authors who wish their extensions to be portable to Unix or OS/2 should use the Unix prefixes, since the Unix-OS/2 version of `ext()` requires them.
- Wherever possible, shareable images are preferred to object libraries, and object libraries to plain object files. In accordance with VMS naming conventions, `ext()` looks for files named `libshr` and `librtl`; it also looks for `liblib` and `liblib` to accommodate Unix conventions used in some ported software.

- For each library that is found, an appropriate directive for a linker options file is generated. The return values are space-separated strings of these directives, rather than elements used on the linker command line.
- LDLOADLIBS contains both the libraries found based on `$potential_libs` and the CRTs, if any, specified in `Config.pm`. EXTRALIBS contains just those libraries found based on `$potential_libs`. BSLOADLIBS and LD_RUN_PATH are always empty.

In addition, an attempt is made to recognize several common Unix library names, and filter them out or convert them to their VMS equivalents, as appropriate.

In general, the VMS version of `ext()` should properly handle input from extensions originally designed for a Unix or VMS environment. If you encounter problems, or discover cases where the search could be improved, please let us know.

Win32 implementation

The version of `ext()` which is executed under Win32 differs from the Unix-OS/2 version in several respects:

- If `$potential_libs` is empty, the return value will be empty. Otherwise, the libraries specified by `$Config{perllibs}` (see `Config.pm`) will be appended to the list of `$potential_libs`. The libraries will be searched for in the directories specified in `$potential_libs`, `$Config{libpth}`, and in `$Config{installarchlib}/CORE`. For each library that is found, a space-separated list of fully qualified library pathnames is generated.
- Input library and path specifications are accepted with or without the `-l` and `-L` prefixes used by Unix linkers.

An entry of the form `-La:\foo` specifies the `a:\foo` directory to look for the libraries that follow.

An entry of the form `-lfoo` specifies the library `foo`, which may be spelled differently depending on what kind of compiler you are using. If you are using GCC, it gets translated to `libfoo.a`, but for other win32 compilers, it becomes `foo.lib`. If no files are found by those translated names, one more attempt is made to find them using either `foo.a` or `libfoo.lib`, depending on whether GCC or some other win32 compiler is being used, respectively.

If neither the `-L` or `-l` prefix is present in an entry, the entry is considered a directory to search if it is in fact a directory, and a library to search for otherwise. The `$Config{lib_ext}` suffix will be appended to any entries that are not directories and don't already have the suffix.

Note that the `-L` and `-l` prefixes are **not required**, but authors who wish their extensions to be portable to Unix or OS/2 should use the prefixes, since the Unix-OS/2 version of `ext()` requires them.

- Entries cannot be plain object files, as many Win32 compilers will not handle object files in the place of libraries.
- Entries in `$potential_libs` beginning with a colon and followed by alphanumeric characters are treated as flags. Unknown flags will be ignored.

An entry that matches `/:nodefault/i` disables the appending of default libraries found in `$Config{perllibs}` (this should be only needed very rarely).

An entry that matches `/:nosearch/i` disables all searching for the libraries specified after it. Translation of `-Lfoo` and `-lfoo` still happens as appropriate (depending on compiler being used, as reflected by `$Config{cc}`), but the entries are not verified to be valid files or directories.

An entry that matches `/:search/i` reenables searching for the libraries specified after it. You can put it at the end to enable searching for default libraries specified by `$Config{perllibs}`.

- The libraries specified may be a mixture of static libraries and import libraries (to link with DLLs). Since both kinds are used pretty transparently on the Win32 platform, we do not attempt to distinguish between them.

- LDLOADLIBS and EXTRALIBS are always identical under Win32, and BSLOADLIBS and LD_RUN_PATH are always empty (this may change in future).
- You must make sure that any paths and path components are properly surrounded with double-quotes if they contain spaces. For example, `$potential_libs` could be (literally):

```
"-Lc:\Program Files\vc\lib" msvcrt.lib "la test\foo bar.lib"
```

Note how the first and last entries are protected by quotes in order to protect the spaces.

- Since this module is most often used only indirectly from extension `Makefile.PL` files, here is an example `Makefile.PL` entry to add a library to the build process for an extension:

```
LIBS => ['-lg1']
```

When using GCC, that entry specifies that MakeMaker should first look for `libg1.a` (followed by `g1.a`) in all the locations specified by `$Config{libpth}`.

When using a compiler other than GCC, the above entry will search for `g1.lib` (followed by `libg1.lib`).

If the library happens to be in a location not in `$Config{libpth}`, you need:

```
LIBS => ['-Lc:\gllibs -lg1']
```

Here is a less often used example:

```
LIBS => ['-lg1', ':nosearch -Ld:\mesalibs -lmesa -luser32']
```

This specifies a search for library `g1` as before. If that search fails to find the library, it looks at the next item in the list. The `:nosearch` flag will prevent searching for the libraries that follow, so it simply returns the value as `-Ld:\mesalibs -lmesa -luser32`, since GCC can use that value as is with its linker.

When using the Visual C compiler, the second item is returned as `-libpath:d:\mesalibs mesa.lib user32.lib`.

When using the Borland compiler, the second item is returned as `-Ld:\mesalibs mesa.lib user32.lib`, and MakeMaker takes care of moving the `-Ld:\mesalibs` to the correct place in the linker command line.

SEE ALSO

[*ExtUtils::MakeMaker*](#)

NAME

ExtUtils::MakeMaker – create an extension Makefile

SYNOPSIS

```
use ExtUtils::MakeMaker;

WriteMakefile( ATTRIBUTE => VALUE [, ...] );
```

DESCRIPTION

This utility is designed to write a Makefile for an extension module from a Makefile.PL. It is based on the Makefile.SH model provided by Andy Dougherty and the perl5-porters.

It splits the task of generating the Makefile into several subroutines that can be individually overridden. Each subroutine returns the text it wishes to have written to the Makefile.

MakeMaker is object oriented. Each directory below the current directory that contains a Makefile.PL is treated as a separate object. This makes it possible to write an unlimited number of Makefiles with a single invocation of `WriteMakefile()`.

How To Write A Makefile.PL

The short answer is: Don't.

```
Always begin with h2xs.
Always begin with h2xs!
ALWAYS BEGIN WITH H2XS!
```

even if you're not building around a header file, and even if you don't have an XS component.

Run `h2xs(1)` before you start thinking about writing a module. For so called pm-only modules that consist of *.pm files only, `h2xs` has the `-X` switch. This will generate dummy files of all kinds that are useful for the module developer.

The medium answer is:

```
use ExtUtils::MakeMaker;
WriteMakefile( NAME => "Foo::Bar" );
```

The long answer is the rest of the manpage :-)

Default Makefile Behaviour

The generated Makefile enables the user of the extension to invoke

```
perl Makefile.PL # optionally "perl Makefile.PL verbose"
make
make test        # optionally set TEST_VERBOSE=1
make install     # See below
```

The Makefile to be produced may be altered by adding arguments of the form `KEY=VALUE`. E.g.

```
perl Makefile.PL PREFIX=/tmp/myperl5
```

Other interesting targets in the generated Makefile are

```
make config      # to check if the Makefile is up-to-date
make clean       # delete local temp files (Makefile gets renamed)
make realclean   # delete derived files (including ./blib)
make ci          # check in all the files in the MANIFEST file
make dist        # see below the Distribution Support section
```

make test

MakeMaker checks for the existence of a file named *test.pl* in the current directory and if it exists it execute the script with the proper set of `perl -I` options.

MakeMaker also checks for any files matching `glob("t/*.t")`. It will execute all matching files in alphabetical order via the [Test::Harness](#) module with the `-I` switches set correctly.

If you'd like to see the raw output of your tests, set the `TEST_VERBOSE` variable to true.

```
make test TEST_VERBOSE=1
```

make testdb

A useful variation of the above is the target `testdb`. It runs the test under the Perl debugger (see [perldebug](#)). If the file *test.pl* exists in the current directory, it is used for the test.

If you want to debug some other testfile, set the `TEST_FILE` variable thusly:

```
make testdb TEST_FILE=t/mytest.t
```

By default the debugger is called using `-d` option to perl. If you want to specify some other option, set the `TESTDB_SW` variable:

```
make testdb TESTDB_SW=-Dx
```

make install

`make` alone puts all relevant files into directories that are named by the macros `INST_LIB`, `INST_ARCHLIB`, `INST_SCRIPT`, `INST_MAN1DIR` and `INST_MAN3DIR`. All these default to something below `./lib` if you are *not* building below the perl source directory. If you *are* building below the perl source, `INST_LIB` and `INST_ARCHLIB` default to `../lib`, and `INST_SCRIPT` is not defined.

The *install* target of the generated Makefile copies the files found below each of the `INST_*` directories to their `INSTALL*` counterparts. Which counterparts are chosen depends on the setting of `INSTALLDIRS` according to the following table:

		INSTALLDIRS set to	
		perl	site vendor
	PREFIX	SITEPREFIX	VENDORPREFIX
<code>INST_ARCHLIB</code>	<code>INSTALLARCHLIB</code>	<code>INSTALLSITEARCH</code>	<code>INSTALLVENDORARCH</code>
<code>INST_LIB</code>	<code>INSTALLPRIVLIB</code>	<code>INSTALLSITELIB</code>	<code>INSTALLVENDORLIB</code>
<code>INST_BIN</code>	<code>INSTALLBIN</code>	<code>INSTALLSITEBIN</code>	<code>INSTALLVENDORBIN</code>
<code>INST_SCRIPT</code>	<code>INSTALLSCRIPT</code>	<code>INSTALLSCRIPT</code>	<code>INSTALLSCRIPT</code>
<code>INST_MAN1DIR</code>	<code>INSTALLMAN1DIR</code>	<code>INSTALLSITEMAN1DIR</code>	<code>INSTALLVENDORMAN1DIR</code>
<code>INST_MAN3DIR</code>	<code>INSTALLMAN3DIR</code>	<code>INSTALLSITEMAN3DIR</code>	<code>INSTALLVENDORMAN3DIR</code>

The `INSTALL...` macros in turn default to their `%Config` (`$Config{installprivlib}`, `$Config{installarchlib}`, etc.) counterparts.

You can check the values of these variables on your system with

```
perl '-V:install.*'
```

And to check the sequence in which the library directories are searched by perl, run

```
perl -le 'print join $/, @INC'
```

PREFIX and LIB attribute

`PREFIX` and `LIB` can be used to set several `INSTALL*` attributes in one go. The quickest way to install a module in a non-standard place might be

```
perl Makefile.PL PREFIX=~
```

This will install all files in the module under your home directory, with man pages and libraries going into an appropriate place (usually `~/man` and `~/lib`).

Another way to specify many `INSTALL` directories with a single parameter is `LIB`.

```
perl Makefile.PL LIB=~lib
```

This will install the module's architecture-independent files into `~/lib`, the architecture-dependent files into `~/lib/$archname`.

Note, that in both cases the tilde expansion is done by MakeMaker, not by perl by default, nor by make.

Conflicts between parameters LIB, PREFIX and the various INSTALL* arguments are resolved so that:

- setting LIB overrides any setting of INSTALLPRIVLIB, INSTALLARCHLIB, INSTALLSITELIB, INSTALLSITEARCH (and they are not affected by PREFIX);
- without LIB, setting PREFIX replaces the initial `$Config{prefix}` part of those INSTALL* arguments, even if the latter are explicitly set (but are set to still start with `$Config{prefix}`).

If the user has superuser privileges, and is not working on AFS or relatives, then the defaults for INSTALLPRIVLIB, INSTALLARCHLIB, INSTALLSCRIPT, etc. will be appropriate, and this incantation will be the best:

```
perl Makefile.PL;
make;
make test
make install
```

make install per default writes some documentation of what has been done into the file `$(INSTALLARCHLIB)/perllocal.pod`. This feature can be bypassed by calling `make pure_install`.

AFS users

will have to specify the installation directories as these most probably have changed since perl itself has been installed. They will have to do this by calling

```
perl Makefile.PL INSTALLSITELIB=/afs/here/today \
    INSTALLSCRIPT=/afs/there/now INSTALLMAN3DIR=/afs/for/manpages
make
```

Be careful to repeat this procedure every time you recompile an extension, unless you are sure the AFS installation directories are still valid.

Static Linking of a new Perl Binary

An extension that is built with the above steps is ready to use on systems supporting dynamic loading. On systems that do not support dynamic loading, any newly created extension has to be linked together with the available resources. MakeMaker supports the linking process by creating appropriate targets in the Makefile whenever an extension is built. You can invoke the corresponding section of the makefile with

```
make perl
```

That produces a new perl binary in the current directory with all extensions linked in that can be found in INST_ARCHLIB, SITELIBEXP, and PERL_ARCHLIB. To do that, MakeMaker writes a new Makefile, on UNIX, this is called `Makefile.aperl` (may be system dependent). If you want to force the creation of a new perl, it is recommended, that you delete this `Makefile.aperl`, so the directories are searched-through for linkable libraries again.

The binary can be installed into the directory where perl normally resides on your machine with

```
make inst_perl
```

To produce a perl binary with a different name than `perl`, either say

```
perl Makefile.PL MAP_TARGET=myperl
make myperl
make inst_perl
```

or say

```
perl Makefile.PL
make myperl MAP_TARGET=myperl
```

```
make inst_perl MAP_TARGET=myperl
```

In any case you will be prompted with the correct invocation of the `inst_perl` target that installs the new binary into `INSTALLBIN`.

`make inst_perl` per default writes some documentation of what has been done into the file `$(INSTALLARCHLIB)/perllocal.pod`. This can be bypassed by calling `make pure_inst_perl`.

Warning: the `inst_perl` target will most probably overwrite your existing perl binary. Use with care!

Sometimes you might want to build a statically linked perl although your system supports dynamic loading. In this case you may explicitly set the `linktype` with the invocation of the `Makefile.PL` or `make`:

```
perl Makefile.PL LINKTYPE=static      # recommended
```

or

```
make LINKTYPE=static                  # works on most systems
```

Determination of Perl Library and Installation Locations

MakeMaker needs to know, or to guess, where certain things are located. Especially `INST_LIB` and `INST_ARCHLIB` (where to put the files during the `make(1)` run), `PERL_LIB` and `PERL_ARCHLIB` (where to read existing modules from), and `PERL_INC` (header files and `libperl*.*`).

Extensions may be built either using the contents of the perl source directory tree or from the installed perl library. The recommended way is to build extensions after you have run ‘`make install`’ on perl itself. You can do that in any directory on your hard disk that is not below the perl source tree. The support for extensions below the `ext` directory of the perl distribution is only good for the standard extensions that come with perl.

If an extension is being built below the `ext/` directory of the perl source then MakeMaker will set `PERL_SRC` automatically (e.g., `../..`). If `PERL_SRC` is defined and the extension is recognized as a standard extension, then other variables default to the following:

```
PERL_INC      = PERL_SRC
PERL_LIB      = PERL_SRC/lib
PERL_ARCHLIB  = PERL_SRC/lib
INST_LIB      = PERL_LIB
INST_ARCHLIB  = PERL_ARCHLIB
```

If an extension is being built away from the perl source then MakeMaker will leave `PERL_SRC` undefined and default to using the installed copy of the perl library. The other variables default to the following:

```
PERL_INC      = $archlibexp/CORE
PERL_LIB      = $privlibexp
PERL_ARCHLIB  = $archlibexp
INST_LIB      = ./blib/lib
INST_ARCHLIB  = ./blib/arch
```

If perl has not yet been installed then `PERL_SRC` can be defined on the command line as shown in the previous section.

Which architecture dependent directory?

If you don’t want to keep the defaults for the `INSTALL*` macros, MakeMaker helps you to minimize the typing needed: the usual relationship between `INSTALLPRIVLIB` and `INSTALLARCHLIB` is determined by `Configure` at perl compilation time. MakeMaker supports the user who sets `INSTALLPRIVLIB`. If `INSTALLPRIVLIB` is set, but `INSTALLARCHLIB` not, then MakeMaker defaults the latter to be the same subdirectory of `INSTALLPRIVLIB` as `Configure` decided for the counterparts in `%Config`, otherwise it defaults to `INSTALLPRIVLIB`. The same relationship holds for `INSTALLSITELIB` and `INSTALLSITEARCH`.

MakeMaker gives you much more freedom than needed to configure internal variables and get different results. It is worth to mention, that `make(1)` also lets you configure most of the variables that are used in the

Makefile. But in the majority of situations this will not be necessary, and should only be done if the author of a package recommends it (or you know what you're doing).

Using Attributes and Parameters

The following attributes may be specified as arguments to `WriteMakefile()` or as `NAME=VALUE` pairs on the command line.

ABSTRACT

One line description of the module. Will be included in PPD file.

ABSTRACT_FROM

Name of the file that contains the package description. MakeMaker looks for a line in the POD matching `/($\$package$ \s-\s)(.*)/`. This is typically the first line in the `"=head1 NAME"` section. `$2` becomes the abstract.

AUTHOR

String containing name (and email address) of package author(s). Is used in PPD (Perl Package Description) files for PPM (Perl Package Manager).

BINARY_LOCATION

Used when creating PPD files for binary packages. It can be set to a full or relative path or URL to the binary archive for a particular architecture. For example:

```
perl Makefile.PL BINARY_LOCATION=x86/Agent.tar.gz
```

builds a PPD package that references a binary of the `Agent` package, located in the `x86` directory relative to the PPD itself.

C Ref to array of *.c file names. Initialised from a directory scan and the values portion of the XS attribute hash. This is not currently used by MakeMaker but may be handy in Makefile.PLs.

CCFLAGS

String that will be included in the compiler call command line between the arguments `INC` and `OPTIMIZE`.

CONFIG

Arrayref. E.g. `[qw(archname manext)]` defines `ARCHNAME` & `MANEXT` from `config.sh`. MakeMaker will add to `CONFIG` the following values anyway: `ar cc ccldflags ccldflags dlexit dlsrc ld ldldflags ldflags libc lib_ext obj_ext ranlib sitelibexp sitearchexp` so

CONFIGURE

CODE reference. The subroutine should return a hash reference. The hash may contain further attributes, e.g. `{LIBS => ...}`, that have to be determined by some evaluation method.

DEFINE

Something like `"-DHAVE_UNISTD_H"`

DIR

Ref to array of subdirectories containing Makefile.PLs e.g. `['sdbm']` in `ext/SDBM_File`

DISTNAME

Your name for distributing the package (by tar file). This defaults to `NAME` above.

DL_FUNCS

Hashref of symbol names for routines to be made available as universal symbols. Each key/value pair consists of the package name and an array of routine names in that package. Used only under AIX, OS/2, VMS and Win32 at present. The routine names supplied will be expanded in the same way as XSUB names are expanded by the `XS()` macro. Defaults to

```
{ "$ (NAME) " => [ "boot_$ (NAME) " ] }
```

e.g.

```
{ "RPC" => [qw( boot_rpcb rpcb_gettime getnetconfignt )],
  "NetconfigPtr" => [ 'DESTROY' ] }
```

Please see the [ExtUtils::Mksymlists](#) documentation for more information about the DL_FUNCS, DL_VARS and FUNCLIST attributes.

DL_VARS

Array of symbol names for variables to be made available as universal symbols. Used only under AIX, OS/2, VMS and Win32 at present. Defaults to []. (e.g. [qw(Foo_version Foo_numstreams Foo_tree)])

EXCLUDE_EXT

Array of extension names to exclude when doing a static build. This is ignored if INCLUDE_EXT is present. Consult INCLUDE_EXT for more details. (e.g. [qw(Socket POSIX)])

This attribute may be most useful when specified as a string on the command line: perl Makefile.PL EXCLUDE_EXT='Socket Safe'

EXE_FILES

Ref to array of executable files. The files will be copied to the INST_SCRIPT directory. Make realclean will delete them from there again.

FIRST_MAKEFILE

The name of the Makefile to be produced. Defaults to the contents of MAKEFILE, but can be overridden. This is used for the second Makefile that will be produced for the MAP_TARGET.

FULLPERL

Perl binary able to run this extension, load XS modules, etc...

FULLPERLRUN

Like PERLRUN, except it uses FULLPERL.

FULLPERLRUNINST

Like PERLRUNINST, except it uses FULLPERL.

FUNCLIST

This provides an alternate means to specify function names to be exported from the extension. Its value is a reference to an array of function names to be exported by the extension. These names are passed through unaltered to the linker options file.

H Ref to array of *.h file names. Similar to C.

IMPORTS

This attribute is used to specify names to be imported into the extension. Takes a hash ref.

It is only used on OS/2 and Win32.

INC

Include file dirs eg: "-I/usr/5include -I/path/to/inc"

INCLUDE_EXT

Array of extension names to be included when doing a static build. MakeMaker will normally build with all of the installed extensions when doing a static build, and that is usually the desired behavior. If INCLUDE_EXT is present then MakeMaker will build only with those extensions which are explicitly mentioned. (e.g. [qw(Socket POSIX)])

It is not necessary to mention DynaLoader or the current extension when filling in INCLUDE_EXT. If the INCLUDE_EXT is mentioned but is empty then only DynaLoader and the current extension will be included in the build.

This attribute may be most useful when specified as a string on the command line: `perl Makefile.PL INCLUDE_EXT='POSIX Socket Devel::Peek'`

INSTALLARCHLIB

Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to perl.

INSTALLBIN

Directory to install binary files (e.g. tkperl) into if INSTALLDIRS=perl.

INSTALLDIRS

Determines which of the sets of installation directories to choose: perl, site or vendor. Defaults to site.

INSTALLMAN1DIR

INSTALLMAN3DIR

These directories get the man pages at 'make install' time if INSTALLDIRS=perl. Defaults to `$Config{installman*dir}`.

If set to 'none', no man pages will be installed.

INSTALLPRIVLIB

Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to perl.

Defaults to `$Config{installprivlib}`.

INSTALLSCRIPT

Used by 'make install' which copies files from INST_SCRIPT to this directory.

INSTALLSITEARCH

Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to site (default).

INSTALLSITEBIN

Used by 'make install', which copies files from INST_BIN to this directory if INSTALLDIRS is set to site (default).

INSTALLSITELIB

Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to site (default).

INSTALLSITEMAN1DIR

INSTALLSITEMAN3DIR

These directories get the man pages at 'make install' time if INSTALLDIRS=site (default). Defaults to `$(SITEPREFIX)/man/man$(MAN*EXT)`.

If set to 'none', no man pages will be installed.

INSTALLVENDORARCH

Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to vendor.

INSTALLVENDORBIN

Used by 'make install', which copies files from INST_BIN to this directory if INSTALLDIRS is set to vendor.

INSTALLVENDORLIB

Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to vendor.

INSTALLVENDORMAN1DIR

INSTALLVENDORMAN3DIR

These directories get the man pages at 'make install' time if `INSTALLDIRS=vendor`. Defaults to `$(VENDORPREFIX) /man/man$(MAN*EXT)`.

If set to 'none', no man pages will be installed.

INST_ARCHLIB

Same as `INST_LIB` for architecture dependent files.

INST_BIN

Directory to put real binary files during 'make'. These will be copied to `INSTALLBIN` during 'make install'

INST_LIB

Directory where we put library files of this extension while building it.

INST_MAN1DIR

Directory to hold the man pages at 'make' time

INST_MAN3DIR

Directory to hold the man pages at 'make' time

INST_SCRIPT

Directory, where executable files should be installed during 'make'. Defaults to `"/blib/script"`, just to have a dummy location during testing. `make install` will copy the files in `INST_SCRIPT` to `INSTALLSCRIPT`.

LDDLFLAGS

Any special flags that might need to be passed to `ld` to create a shared library suitable for dynamic loading. It is up to the makefile to use it. (See [lddlflags](#))

Defaults to `$Config{lddlflags}`.

LDFROM

Defaults to `"$(OBJECT) "` and is used in the `ld` command to specify what files to link/load from (also see `dynamic_lib` below for how to specify `ld` flags)

LIB

`LIB` should only be set at `perl Makefile.PL` time but is allowed as a `MakeMaker` argument. It has the effect of setting both `INSTALLPRIVLIB` and `INSTALLSITELIB` to that value regardless any explicit setting of those arguments (or of `PREFIX`). `INSTALLARCHLIB` and `INSTALLSITEARCH` are set to the corresponding architecture subdirectory.

LIBPERL_A

The filename of the perl library that will be used together with this extension. Defaults to `libperl.a`.

LIBS

An anonymous array of alternative library specifications to be searched for (in order) until at least one library is found. E.g.

```
'LIBS' => ["-lgdbm", "-ldbm -lfoo", "-L/path -ldbm.nfs"]
```

Mind, that any element of the array contains a complete set of arguments for the `ld` command. So do not specify

```
'LIBS' => ["-ltcl", "-ltk", "-lX11"]
```

See `ODBM_File/Makefile.PL` for an example, where an array is needed. If you specify a scalar as in

```
'LIBS' => "-ltcl -ltk -lX11"
```

MakeMaker will turn it into an array with one element.

LINKTYPE

'static' or 'dynamic' (default unless `usedl=undef` in `config.sh`). Should only be used to force static linking (also see `linkext` below).

MAKEAPERL

Boolean which tells MakeMaker, that it should include the rules to make a perl. This is handled automatically as a switch by MakeMaker. The user normally does not need it.

MAKEFILE

The name of the Makefile to be produced.

MAN1PODS

Hashref of pod-containing files. MakeMaker will default this to all `EXE_FILES` files that include POD directives. The files listed here will be converted to man pages and installed as was requested at Configure time.

MAN3PODS

Hashref that assigns to *.pm and *.pod files the files into which the manpages are to be written. MakeMaker parses all *.pod and *.pm files for POD directives. Files that contain POD will be the default keys of the MAN3PODS hashref. These will then be converted to man pages during make and will be installed during `make install`.

MAP_TARGET

If it is intended, that a new perl binary be produced, this variable may hold a name for that binary. Defaults to perl

MYEXTLIB

If the extension links to a library that it builds set this to the name of the library (see `SDBM_File`)

NAME

Perl module name for this extension (`DBD::Oracle`). This will default to the directory name but should be explicitly defined in the `Makefile.PL`.

NEEDS_LINKING

MakeMaker will figure out if an extension contains linkable code anywhere down the directory tree, and will set this variable accordingly, but you can speed it up a very little bit if you define this boolean variable yourself.

NOECHO

Defaults to @. By setting it to an empty string you can generate a Makefile that echos all commands. Mainly used in debugging MakeMaker itself.

NORECURS

Boolean. Attribute to inhibit descending into subdirectories.

NO_VC

In general, any generated Makefile checks for the current version of MakeMaker and the version the Makefile was built under. If `NO_VC` is set, the version check is neglected. Do not write this into your `Makefile.PL`, use it interactively instead.

OBJECT

List of object files, defaults to `'$(BASEEXT) $(OBJ_EXT) '`, but can be a long string containing all object files, e.g. `"tkpBind.o tkpButton.o tkpCanvas.o"`

(Where `BASEEXT` is the last component of `NAME`, and `OBJ_EXT` is `$Config{obj_ext}`.)

OPTIMIZE

Defaults to `-O`. Set it to `-g` to turn debugging on. The flag is passed to subdirectory makes.

PERL

Perl binary for tasks that can be done by `miniperl`

PERL_CORE

Set only when MakeMaker is building the extensions of the Perl core distribution.

PERLMAINCC

The call to the program that is able to compile `perlmain.c`. Defaults to `$(CC)`.

PERL_ARCHLIB

Same as for `PERL_LIB`, but for architecture dependent files.

Used only when MakeMaker is building the extensions of the Perl core distribution (because normally `$(PERL_ARCHLIB)` is automatically in `@INC`, and adding it would get in the way of `PERL5LIB`).

PERL_LIB

Directory containing the Perl library to use.

Used only when MakeMaker is building the extensions of the Perl core distribution (because normally `$(PERL_LIB)` is automatically in `@INC`, and adding it would get in the way of `PERL5LIB`).

PERL_MALLOC_OK

defaults to 0. Should be set to `TRUE` if the extension can work with the memory allocation routines substituted by the Perl `malloc()` subsystem. This should be applicable to most extensions with exceptions of those

- with bugs in memory allocations which are caught by Perl's `malloc()`;
- which interact with the memory allocator in other ways than via `malloc()`, `realloc()`, `free()`, `calloc()`, `sbrk()` and `brk()`;
- which rely on special alignment which is not provided by Perl's `malloc()`.

NOTE. Negligence to set this flag in *any one* of loaded extension nullifies many advantages of Perl's `malloc()`, such as better usage of system resources, error detection, memory usage reporting, catchable failure of memory allocations, etc.

PERLRUN

Use this instead of `$(PERL)` when you wish to run `perl`. It will set up extra necessary flags for you.

PERLRUNINST

Use this instead of `$(PERL)` when you wish to run `perl` to work with modules. It will add things like `-I$(INST_ARCH)` and other necessary flags so `perl` can see the modules you're about to install.

PERL_SRC

Directory containing the Perl source code (use of this should be avoided, it may be undefined)

PERM_RW

Desired permission for read/writable files. Defaults to `644`. See also [perm_rw](#).

PERM_RWX

Desired permission for executable files. Defaults to `755`. See also [perm_rwx](#).

PL_FILES

Ref to hash of files to be processed as perl programs. MakeMaker will default to any found `*.PL` file (except `Makefile.PL`) being keys and the basename of the file being the value. E.g.

```
{ 'foobar.PL' => 'foobar' }
```

The *.PL files are expected to produce output to the target files themselves. If multiple files can be generated from the same *.PL file then the value in the hash can be a reference to an array of target file names. E.g.

```
{ 'foobar.PL' => ['foobar1', 'foobar2'] }
```

PM

Hashref of .pm files and *.pl files to be installed. e.g.

```
{ 'name_of_file.pm' => '$(INST_LIBDIR)/install_as.pm' }
```

By default this will include *.pm and *.pl and the files found in the PMLIBDIRS directories. Defining PM in the Makefile.PL will override PMLIBDIRS.

PMLIBDIRS

Ref to array of subdirectories containing library files. Defaults to ['lib', \$(BASEEXT)]. The directories will be scanned and *any* files they contain will be installed in the corresponding location in the library. A `libscan()` method can be used to alter the behaviour. Defining PM in the Makefile.PL will override PMLIBDIRS.

(Where BASEEXT is the last component of NAME.)

PM_FILTER

A filter program, in the traditional Unix sense (input from stdin, output to stdout) that is passed on each .pm file during the build (in the `pm_to_blib()` phase). It is empty by default, meaning no filtering is done.

Great care is necessary when defining the command if quoting needs to be done. For instance, you would need to say:

```
{ 'PM_FILTER' => 'grep -v \\"^\\#\\\"' }
```

to remove all the leading comments on the fly during the build. The extra `\\` are necessary, unfortunately, because this variable is interpolated within the context of a Perl program built on the command line, and double quotes are what is used with the `-e` switch to build that command line. The `#` is escaped for the Makefile, since what is going to be generated will then be:

```
PM_FILTER = grep -v \\"^\\#\\\"
```

Without the `\\` before the `#`, we'd have the start of a Makefile comment, and the macro would be incorrectly defined.

POLLUTE

Release 5.005 grandfathered old global symbol names by providing preprocessor macros for extension source compatibility. As of release 5.6, these preprocessor definitions are not available by default. The POLLUTE flag specifies that the old names should still be defined:

```
perl Makefile.PL POLLUTE=1
```

Please inform the module author if this is necessary to successfully install a module under 5.6 or later.

PPM_INSTALL_EXEC

Name of the executable used to run PPM_INSTALL_SCRIPT below. (e.g. perl)

PPM_INSTALL_SCRIPT

Name of the script that gets executed by the Perl Package Manager after the installation of a package.

PREFIX

This overrides all the default install locations. Man pages, libraries, scripts, etc... MakeMaker will try to make an educated guess about where to place things under the new PREFIX based on your Config defaults. Failing that, it will fall back to a structure which should be sensible for your platform.

If you specify LIB or any INSTALL* variables they will not be effected by the PREFIX.

Defaults to `$Config{installprefixexp}`.

PREREQ_FATAL

Bool. If this parameter is true, failing to have the required modules (or the right versions thereof) will be fatal. perl Makefile.PL will die with the proper message.

Note: see [Test::Harness](#) for a shortcut for stopping tests early if you are missing dependencies.

Do *not* use this parameter for simple requirements, which could be resolved at a later time, e.g. after an unsuccessful **make test** of your module.

It is *extremely* rare to have to use PREREQ_FATAL at all!

PREREQ_PM

Hashref: Names of modules that need to be available to run this extension (e.g. Fcntl for SDBM_File) are the keys of the hash and the desired version is the value. If the required version number is 0, we only check if any version is installed already.

PREREQ_PRINT

Bool. If this parameter is true, the prerequisites will be printed to stdout and MakeMaker will exit. The output format is

```
$PREREQ_PM = {
    'A::B' = Vers1,
    'C::D' = Vers2,
    ...
};
```

PRINT_PREREQ

RedHatism for PREREQ_PRINT. The output format is different, though:

```
perl (A::B) >=Vers1 perl (C::D) >=Vers2 ...
```

SITEPREFIX

Like PREFIX, but only for the site install locations.

Defaults to PREFIX (if set) or `$Config{siteprefixexp}`. Perls prior to 5.6.0 didn't have an explicit siteprefix in the Config. In those cases `$Config{installprefix}` will be used.

SKIP

Arrayref. E.g. `[qw(name1 name2)]` skip (do not write) sections of the Makefile. Caution! Do not use the SKIP attribute for the negligible speedup. It may seriously damage the resulting Makefile. Only use it if you really need it.

TYPEMAPS

Ref to array of typemap file names. Use this when the typemaps are in some directory other than the current directory or when they are not named **typemap**. The last typemap in the list takes precedence. A typemap in the current directory has highest precedence, even if it isn't listed in TYPEMAPS. The default system typemap has lowest precedence.

VENDORPREFIX

Like PREFIX, but only for the vendor install locations.

Defaults to PREFIX (if set) or `$Config{vendorprefixexp}`

VERBINST

If true, make install will be verbose

VERSION

Your version number for distributing the package. This defaults to 0.1.

VERSION_FROM

Instead of specifying the VERSION in the Makefile.PL you can let MakeMaker parse a file to determine the version number. The parsing routine requires that the file named by VERSION_FROM contains one single line to compute the version number. The first line in the file that contains the regular expression

```
/([\$*])(([\w\:\']*)\bVERSION)\b.*\s=/
```

will be evaluated with `eval()` and the value of the named variable **after** the `eval()` will be assigned to the VERSION attribute of the MakeMaker object. The following lines will be parsed o.k.:

```
$VERSION = '1.00';
*VERSION = \ '1.01';
( $VERSION ) = '$Revision: 1.63 $' =~ /\$Revision:\s+([^\s]+)/;
$FOO::VERSION = '1.10';
*FOO::VERSION = \ '1.11';
our $VERSION = 1.2.3;          # new for perl5.6.0
```

but these will fail:

```
my $VERSION = '1.01';
local $VERSION = '1.02';
local $FOO::VERSION = '1.30';
```

(Putting `my` or `local` on the preceding line will work o.k.)

The file named in VERSION_FROM is not added as a dependency to Makefile. This is not really correct, but it would be a major pain during development to have to rewrite the Makefile for any smallish change in that file. If you want to make sure that the Makefile contains the correct VERSION macro after any change of the file, you would have to do something like

```
depend => { Makefile => '$(VERSION_FROM)' }
```

See attribute `depend` below.

XS

Hashref of .xs files. MakeMaker will default this. e.g.

```
{ 'name_of_file.xs' => 'name_of_file.c' }
```

The .c files will automatically be included in the list of files deleted by a `make clean`.

XSOPT

String of options to pass to `xsubpp`. This might include `-C++` or `-extern`. Do not include `typemaps` here; the `TYPEMAP` parameter exists for that purpose.

XSPROTOARG

May be set to an empty string, which is identical to `-prototypes`, or `-noprotoypes`. See the `xsubpp` documentation for details. MakeMaker defaults to the empty string.

XS_VERSION

Your version number for the .xs file of this package. This defaults to the value of the VERSION attribute.

Additional lowercase attributes

can be used to pass parameters to the methods which implement that part of the Makefile.

clean

```
{FILES => "*.xyz foo"}
```

depend

```
{ANY_TARGET => ANY_DEPENDENCY, ...}
```

(ANY_TARGET must not be given a double-colon rule by MakeMaker.)

dist

```
{TARFLAGS => 'cvfF', COMPRESS => 'gzip', SUFFIX => '.gz',
SHAR => 'shar -m', DIST_CP => 'ln', ZIP => '/bin/zip',
ZIPFLAGS => '-rl', DIST_DEFAULT => 'private tardist' }
```

If you specify COMPRESS, then SUFFIX should also be altered, as it is needed to tell make the target file of the compression. Setting DIST_CP to ln can be useful, if you need to preserve the timestamps on your files. DIST_CP can take the values 'cp', which copies the file, 'ln', which links the file, and 'best' which copies symbolic links and links the rest. Default is 'best'.

dynamic_lib

```
{ARMAYBE => 'ar', OTHERLDFLAGS => '...', INST_DYNAMIC_DEP => '...'}
```

linkext

```
{LINKTYPE => 'static', 'dynamic' or ''}
```

NB: Extensions that have nothing but *.pm files had to say

```
{LINKTYPE => ''}
```

with Pre-5.0 MakeMakers. Since version 5.00 of MakeMaker such a line can be deleted safely. MakeMaker recognizes when there's nothing to be linked.

macro

```
{ANY_MACRO => ANY_VALUE, ...}
```

realclean

```
{FILES => '$(INST_ARCHAUTODIR)/*.xyz'}
```

test

```
{TESTS => 't/*.t'}
```

tool_autosplit

```
{MAXLEN => 8}
```

Overriding MakeMaker Methods

If you cannot achieve the desired Makefile behaviour by specifying attributes you may define private subroutines in the Makefile.PL. Each subroutine returns the text it wishes to have written to the Makefile. To override a section of the Makefile you can either say:

```
sub MY::c_o { "new literal text" }
```

or you can edit the default by saying something like:

```
package MY; # so that "SUPER" works right
sub c_o {
    my $inherited = shift->SUPER::c_o(@_);
    $inherited =~ s/old text/new text/;
    $inherited;
}
```

If you are running experiments with embedding perl as a library into other applications, you might find MakeMaker is not sufficient. You'd better have a look at ExtUtils::Embed which is a collection of utilities

for embedding.

If you still need a different solution, try to develop another subroutine that fits your needs and submit the diffs to makemaker@perl.org

For a complete description of all MakeMaker methods see [ExtUtils::MM_Unix](#).

Here is a simple example of how to add a new target to the generated Makefile:

```
sub MY::postamble {
    return <<'MAKE_FRAG';
    $(MYEXTLIB): sdbm/Makefile
        cd sdbm && $(MAKE) all

    MAKE_FRAG
}
```

The End Of Cargo Cult Programming

`WriteMakefile()` now does some basic sanity checks on its parameters to protect against typos and malformed values. This means some things which happened to work in the past will now throw warnings and possibly produce internal errors.

Some of the most common mistakes:

```
<MAN3PODS = ''
```

This is commonly used to suppress the creation of man pages. `MAN3PODS` takes a hash ref not a string, but the above worked by accident in old versions of MakeMaker.

The correct code is `<MAN3PODS = { }.`

Hintsfile support

`MakeMaker.pm` uses the architecture specific information from `Config.pm`. In addition it evaluates architecture specific hints files in a `hints/` directory. The hints files are expected to be named like their counterparts in `PERL_SRC/hints`, but with an `.pl` file name extension (eg. `next_3_2.pl`). They are simply eval'd by MakeMaker within the `WriteMakefile()` subroutine, and can be used to execute commands as well as to include special variables. The rules which hintsfile is chosen are the same as in `Configure`.

The hintsfile is `eval()`ed immediately after the arguments given to `WriteMakefile` are stuffed into a hash reference `$self` but before this reference becomes blessed. So if you want to do the equivalent to override or create an attribute you would say something like

```
$self->{LIBS} = ['-ldbm -lucb -lc'];
```

Distribution Support

For authors of extensions MakeMaker provides several Makefile targets. Most of the support comes from the `ExtUtils::Manifest` module, where additional documentation can be found.

make distcheck

reports which files are below the build directory but not in the MANIFEST file and vice versa. (See `ExtUtils::Manifest::fullcheck()` for details)

make skipcheck

reports which files are skipped due to the entries in the MANIFEST.SKIP file (See `ExtUtils::Manifest::skipcheck()` for details)

make distclean

does a `realclean` first and then the `distcheck`. Note that this is not needed to build a new distribution as long as you are sure that the MANIFEST file is ok.

make manifest

rewrites the MANIFEST file, adding all remaining files found (See `ExtUtils::Manifest::mkmanifest()` for details)

make distdir

Copies all the files that are in the MANIFEST file to a newly created directory with the name `$(DISTNAME) - $(VERSION)`. If that directory exists, it will be removed first.

make disttest

Makes a distdir first, and runs a `perl Makefile.PL`, a `make`, and a `make test` in that directory.

make tardist

First does a `distdir`. Then a command `$(PREOP)` which defaults to a null command, followed by `$(TOUNIX)`, which defaults to a null command under UNIX, and will convert files in distribution directory to UNIX format otherwise. Next it runs `tar` on that directory into a tarfile and deletes the directory. Finishes with a command `$(POSTOP)` which defaults to a null command.

make dist

Defaults to `$(DIST_DEFAULT)` which in turn defaults to `tardist`.

make uutardist

Runs a `tardist` first and `uuencodes` the tarfile.

make shdist

First does a `distdir`. Then a command `$(PREOP)` which defaults to a null command. Next it runs `shar` on that directory into a sharfile and deletes the intermediate directory again. Finishes with a command `$(POSTOP)` which defaults to a null command. Note: For `shdist` to work properly a `shar` program that can handle directories is mandatory.

make zipdist

First does a `distdir`. Then a command `$(PREOP)` which defaults to a null command. Runs `$(ZIP)` `$(ZIPFLAGS)` on that directory into a zipfile. Then deletes that directory. Finishes with a command `$(POSTOP)` which defaults to a null command.

make ci

Does a `$(CI)` and a `$(RCS_LABEL)` on all files in the MANIFEST file.

Customization of the dist targets can be done by specifying a hash reference to the `dist` attribute of the `WriteMakefile` call. The following parameters are recognized:

<code>CI</code>	<code>('ci -u')</code>
<code>COMPRESS</code>	<code>('gzip --best')</code>
<code>POSTOP</code>	<code>('@ :')</code>
<code>PREOP</code>	<code>('@ :')</code>
<code>TO_UNIX</code>	<code>(depends on the system)</code>
<code>RCS_LABEL</code>	<code>('rcs -q -Nv\$(VERSION_SYM) :')</code>
<code>SHAR</code>	<code>('shar')</code>
<code>SUFFIX</code>	<code>('gz')</code>
<code>TAR</code>	<code>('tar')</code>
<code>TARFLAGS</code>	<code>('cvf')</code>
<code>ZIP</code>	<code>('zip')</code>
<code>ZIPFLAGS</code>	<code>('r')</code>

An example:

```
WriteMakefile( 'dist' => { COMPRESS=>"bzip2", SUFFIX=>".bz2" } )
```

Disabling an extension

If some events detected in *Makefile.PL* imply that there is no way to create the Module, but this is a normal state of things, then you can create a *Makefile* which does nothing, but succeeds on all the "usual" build targets. To do so, use

```
ExtUtils::MakeMaker::WriteEmptyMakefile();
```

instead of `WriteMakefile()`.

This may be useful if other modules expect this module to be *built* OK, as opposed to *work* OK (say, this system-dependent module builds in a subdirectory of some other distribution, or is listed as a dependency in a CPAN::Bundle, but the functionality is supported by different means on the current architecture).

ENVIRONMENT

PERL_MM_OPT

Command line options used by `MakeMaker->new()`, and thus by `WriteMakefile()`. The string is split on whitespace, and the result is processed before any actual command line arguments are processed.

PERL_MM_USE_DEFAULT

If set to a true value then MakeMaker's prompt function will always return the default without waiting for user input.

SEE ALSO

`ExtUtils::MM_Unix`, `ExtUtils::Manifest` `ExtUtils::Install`, `ExtUtils::Embed`

AUTHORS

Andy Dougherty <doughera@lafayette.edu>, Andreas König <andreas.koenig@mind.de>, Tim Bunce <timb@cpan.org>. VMS support by Charles Bailey <bailey@newman.upenn.edu>. OS/2 support by Ilya Zakharevich <ilya@math.ohio-state.edu>.

Currently maintained by Michael G Schwern <schwern@pobox.com>

Send patches and ideas to <makemaker@perl.org>.

Send bug reports via <http://rt.cpan.org/>. Please send your generated Makefile along with your report.

For more up-to-date information, see <http://www.makemaker.org>.

NAME

ExtUtils::Manifest – utilities to write and check a MANIFEST file

SYNOPSIS

```
require ExtUtils::Manifest;

ExtUtils::Manifest::mkmanifest;

ExtUtils::Manifest::manicheck;

ExtUtils::Manifest::filecheck;

ExtUtils::Manifest::fullcheck;

ExtUtils::Manifest::skipcheck;

ExtUtils::Manifest::manifind();

ExtUtils::Manifest::maniread($file);

ExtUtils::Manifest::manicopy($read,$target,$show);
```

DESCRIPTION

`mkmanifest()` writes all files in and below the current directory to a file named in the global variable `$ExtUtils::Manifest::MANIFEST` (which defaults to `MANIFEST`) in the current directory. It works similar to

```
find . -print
```

but in doing so checks each line in an existing `MANIFEST` file and includes any comments that are found in the existing `MANIFEST` file in the new one. Anything between white space and an end of line within a `MANIFEST` file is considered to be a comment. Filenames and comments are separated by one or more `TAB` characters in the output. All files that match any regular expression in a file `MANIFEST.SKIP` (if such a file exists) are ignored.

`manicheck()` checks if all the files within a `MANIFEST` in the current directory really do exist. If `MANIFEST` and the tree below the current directory are in sync it exits silently, returning an empty list. Otherwise it returns a list of files which are listed in the `MANIFEST` but missing from the directory, and by default also outputs these names to `STDERR`.

`filecheck()` finds files below the current directory that are not mentioned in the `MANIFEST` file. An optional file `MANIFEST.SKIP` will be consulted. Any file matching a regular expression in such a file will not be reported as missing in the `MANIFEST` file. The list of any extraneous files found is returned, and by default also reported to `STDERR`.

`fullcheck()` does both a `manicheck()` and a `filecheck()`, returning references to two arrays, the first for files `manicheck()` found to be missing, the second for unexpected files found by `filecheck()`.

`skipcheck()` lists all the files that are skipped due to your `MANIFEST.SKIP` file.

`manifind()` returns a hash reference. The keys of the hash are the files found below the current directory.

`maniread($file)` reads a named `MANIFEST` file (defaults to `MANIFEST` in the current directory) and returns a `HASH` reference with files being the keys and comments being the values of the `HASH`. Blank lines and lines which start with `#` in the `MANIFEST` file are discarded.

`manicopy($read,$target,$show)` copies the files that are the keys in the `HASH` `$_$read` to the named target directory. The `HASH` reference `$read` is typically returned by the `maniread()` function. This function is useful for producing a directory tree identical to the intended distribution tree. The third parameter `$show` can be used to specify a different methods of "copying". Valid values are `cp`, which actually copies the files, `ln` which creates hard links, and `best` which mostly links the files but copies any symbolic link to make a tree without any symbolic link. `Best` is the default.

MANIFEST.SKIP

The file MANIFEST.SKIP may contain regular expressions of files that should be ignored by `mkmanifest()` and `filecheck()`. The regular expressions should appear one on each line. Blank lines and lines which start with `#` are skipped. Use `\#` if you need a regular expression to start with a sharp character. A typical example:

```
# Version control files and dirs.
\bRCS\b
\bCVS\b
,v$

# Makemaker generated files and dirs.
^MANIFEST\.
^Makefile$
^blib/
^MakeMaker-\d

# Temp, old and emacs backup files.
~$
\old$
^#.*#$
^\.#
```

If no MANIFEST.SKIP file is found, a default set of skips will be used, similar to the example above. If you want nothing skipped, simply make an empty MANIFEST.SKIP file.

EXPORT_OK

`&mkmanifest`, `&manicheck`, `&filecheck`, `&fullcheck`, `&maniread`, and `&manicopy` are exportable.

GLOBAL VARIABLES

`$ExtUtils::Manifest::MANIFEST` defaults to MANIFEST. Changing it results in both a different MANIFEST and a different MANIFEST.SKIP file. This is useful if you want to maintain different distributions for different audiences (say a user version and a developer version including RCS).

`$ExtUtils::Manifest::Quiet` defaults to 0. If set to a true value, all functions act silently.

`$ExtUtils::Manifest::Debug` defaults to 0. If set to a true value, or if `PERL_MM_MANIFEST_DEBUG` is true, debugging output will be produced.

DIAGNOSTICS

All diagnostic output is sent to STDERR.

Not in MANIFEST: *file*

is reported if a file is found which is not in MANIFEST.

Skipping *file*

is reported if a file is skipped due to an entry in MANIFEST.SKIP.

No such file: *file*

is reported if a file mentioned in a MANIFEST file does not exist.

MANIFEST: *\$!*

is reported if MANIFEST could not be opened.

Added to MANIFEST: *file*

is reported by `mkmanifest()` if `$Verbose` is set and a file is added to MANIFEST. `$Verbose` is set to 1 by default.

ENVIRONMENT**PERL_MM_MANIFEST_DEBUG**

Turns on debugging

SEE ALSO

[ExtUtils::MakeMaker](#) which has handy targets for most of the functionality.

AUTHOR

Andreas Koenig <andreas.koenig@anima.de>

NAME

ExtUtils::Mkbootstrap – make a bootstrap file for use by DynaLoader

SYNOPSIS

Mkbootstrap

DESCRIPTION

Mkbootstrap typically gets called from an extension Makefile.

There is no *.bs file supplied with the extension. Instead, there may be a *_BS file which has code for the special cases, like posix for berkeley db on the NeXT.

This file will get parsed, and produce a maybe empty @DynaLoader::dl_resolve_using array for the current architecture. That will be extended by \$BSLOADLIBS, which was computed by ExtUtils::Liblist::ext(). If this array still is empty, we do nothing, else we write a .bs file with an @DynaLoader::dl_resolve_using array.

The *_BS file can put some code into the generated *.bs file by placing it in \$bscode. This is a handy ‘escape’ mechanism that may prove useful in complex situations.

If @DynaLoader::dl_resolve_using contains -L* or -l* entries then Mkbootstrap will automatically add a dl_findfile() call to the generated *.bs file.

NAME

ExtUtils::Mksymlists – write linker options files for dynamic extension

SYNOPSIS

```
use ExtUtils::Mksymlists;
Mksymlists({ NAME      => $name ,
             DL_VARS   => [ $var1, $var2, $var3 ],
             DL_FUNCS => { $pkg1 => [ $func1, $func2 ],
                           $pkg2 => [ $func3 ] } });
```

DESCRIPTION

ExtUtils::Mksymlists produces files used by the linker under some OSs during the creation of shared libraries for dynamic extensions. It is normally called from a MakeMaker-generated Makefile when the extension is built. The linker option file is generated by calling the function `Mksymlists`, which is exported by default from `ExtUtils::Mksymlists`. It takes one argument, a list of key–value pairs, in which the following keys are recognized:

DLBASE

This item specifies the name by which the linker knows the extension, which may be different from the name of the extension itself (for instance, some linkers add an ‘_’ to the name of the extension). If it is not specified, it is derived from the `NAME` attribute. It is presently used only by OS2 and Win32.

DL_FUNCS

This is identical to the `DL_FUNCS` attribute available via MakeMaker, from which it is usually taken. Its value is a reference to an associative array, in which each key is the name of a package, and each value is an a reference to an array of function names which should be exported by the extension. For instance, one might say `DL_FUNCS => { Homer::Iliad => [qw(trojans greeks)], Homer::Odyssey => [qw(travellers family suitors)] }`. The function names should be identical to those in the `XSUB` code; `Mksymlists` will alter the names written to the linker option file to match the changes made by `xsubpp`. In addition, if none of the functions in a list begin with the string `boot_`, `Mksymlists` will add a bootstrap function for that package, just as `xsubpp` does. (If a `boot_<pkg>` function is present in the list, it is passed through unchanged.) If `DL_FUNCS` is not specified, it defaults to the bootstrap function for the extension specified in `NAME`.

DL_VARS

This is identical to the `DL_VARS` attribute available via MakeMaker, and, like `DL_FUNCS`, it is usually specified via MakeMaker. Its value is a reference to an array of variable names which should be exported by the extension.

FILE

This key can be used to specify the name of the linker option file (minus the OS–specific extension), if for some reason you do not want to use the default value, which is the last word of the `NAME` attribute (*e.g.* for `Tk::Canvas`, `FILE` defaults to `Canvas`).

FUNCLIST

This provides an alternate means to specify function names to be exported from the extension. Its value is a reference to an array of function names to be exported by the extension. These names are passed through unaltered to the linker options file. Specifying a value for the `FUNCLIST` attribute suppresses automatic generation of the bootstrap function for the package. To still create the bootstrap name you have to specify the package name in the `DL_FUNCS` hash:

```
Mksymlists({ NAME      => $name ,
             FUNCLIST => [ $func1, $func2 ],
             DL_FUNCS => { $pkg => [ ] } });
```


IMPORTS

This attribute is used to specify names to be imported into the extension. It is currently only used by OS/2 and Win32.

NAME

This gives the name of the extension (*e.g.* Tk : : Canvas) for which the linker option file will be produced.

When calling `Mksymlists`, one should always specify the `NAME` attribute. In most cases, this is all that's necessary. In the case of unusual extensions, however, the other attributes can be used to provide additional information to the linker.

AUTHOR

Charles Bailey <*bailey@newman.upenn.edu*>

REVISION

Last revised 14-Feb-1996, for Perl 5.002.

NAME

ExtUtils::MM – OS adjusted ExtUtils::MakeMaker subclass

SYNOPSIS

```
require ExtUtils::MM;  
my $mm = MM->new(...);
```

DESCRIPTION**FOR INTERNAL USE ONLY**

ExtUtils::MM is a subclass of ExtUtils::MakeMaker which automatically chooses the appropriate OS specific subclass for you (ie. ExtUils::MM_Unix, etc...).

It also provides a convenient alias via the MM class (I didn't want MakeMaker modules outside of ExtUtils/).

This class might turn out to be a temporary solution, but MM won't go away.

NAME

ExtUtils::MM_Any – Platform agnostic MM methods

SYNOPSIS

```
FOR INTERNAL USE ONLY!

package ExtUtils::MM_SomeOS;

# Temporarily, you have to subclass both.  Put MM_Any first.
require ExtUtils::MM_Any;
require ExtUtils::MM_Unix;
@ISA = qw(ExtUtils::MM_Any ExtUtils::Unix);
```

DESCRIPTION**FOR INTERNAL USE ONLY!**

ExtUtils::MM_Any is a superclass for the ExtUtils::MM_* set of modules. It contains methods which are either inherently cross-platform or are written in a cross-platform manner.

Subclass off of ExtUtils::MM_Any *and* ExtUtils::MM_Unix. This is a temporary solution.

THIS MAY BE TEMPORARY!**Inherently Cross-Platform Methods**

These are methods which are by their nature cross-platform and should always be cross-platform.

File::Spec wrappers DEPRECATED

The following methods are deprecated wrappers around File::Spec functions. They exist from before File::Spec did and in fact are from which File::Spec sprang.

They are all deprecated. Please use File::Spec directly.

```
canonpath
catdir
catfile
curdir
file_name_is_absolute
path
rootdir
updir
```

Thought To Be Cross-Platform Methods

These are methods which are thought to be cross-platform by virtue of having been written in a way to avoid incompatibilities.

test_via_harness

```
my $command = $mm->test_via_harness($perl, $tests);
```

Returns a \$command line which runs the given set of \$tests with Test::Harness and the given \$perl.

Used on the t/*.t files.

test_via_script

```
my $command = $mm->test_via_script($perl, $script);
```

Returns a \$command line which just runs a single test without Test::Harness. No checks are done on the results, they're just printed.

Used for test.pl, since they don't always follow Test::Harness formatting.

AUTHOR

Michael G Schwern <schwern@pobox.com with code from ExtUtils::MM_Unix and ExtUtils::MM_Win32.

NAME

ExtUtils::MM_BeOS – methods to override UN*X behaviour in ExtUtils::MakeMaker

SYNOPSIS

```
use ExtUtils::MM_BeOS; # Done internally by ExtUtils::MakeMaker if needed
```

DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

perl_archive

This is internal method that returns path to libperl.a equivalent to be linked to dynamic extensions. UNIX does not have one, but at least BeOS has one.

```
1; __END__
```

NAME

ExtUtils::MM_Cygwin – methods to override UN*X behaviour in ExtUtils::MakeMaker

SYNOPSIS

```
use ExtUtils::MM_Cygwin; # Done internally by ExtUtils::MakeMaker if needed
```

DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there.

canonpath

replaces backslashes with forward ones. then acts as *nixish.

cflags

if configured for dynamic loading, triggers #define EXT in EXTERN.h

manifypods

replaces strings '::' with '.' in man page names

perl_archive

points to libperl.a

NAME

ExtUtils::MM_DOS – DOS specific subclass of ExtUtils::MM_Unix

SYNOPSIS

Don't use this module directly.
Use ExtUtils::MM and let it choose.

DESCRIPTION

This is a subclass of ExtUtils::MM_Unix which contains functionality for DOS.

Unless otherwise stated, it works just like ExtUtils::MM_Unix

Overridden methods

replace_manpage_separator

AUTHOR

Michael G Schwern <schwern@pobox.com with code from ExtUtils::MM_Unix

SEE ALSO

ExtUtils::MM_Unix, ExtUtils::MakeMaker

NAME

ExtUtils::MM_NW5 – methods to override UN*X behaviour in ExtUtils::MakeMaker

SYNOPSIS

```
use ExtUtils::MM_NW5; # Done internally by ExtUtils::MakeMaker if needed
```

DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

constants (o)

Initializes lots of constants and .SUFFIXES and .PHONY

static_lib (o)

dynamic_lib (o)

Defines how to produce the *.so (or equivalent) files.

NAME

ExtUtils::MM_OS2 – methods to override UN*X behaviour in ExtUtils::MakeMaker

SYNOPSIS

```
use ExtUtils::MM_OS2; # Done internally by ExtUtils::MakeMaker if needed
```

DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

METHODS**perl_archive_after**

This is an internal method that returns path to a library which should be put on the linker command line *after* the external libraries to be linked to dynamic extensions. This may be needed if the linker is one-pass, and Perl includes some overrides for C RTL functions, such as `malloc()`.

NAME

ExtUtils::MM_Unix – methods used by ExtUtils::MakeMaker

SYNOPSIS

```
require ExtUtils::MM_Unix;
```

DESCRIPTION

The methods provided by this package are designed to be used in conjunction with ExtUtils::MakeMaker. When MakeMaker writes a Makefile, it creates one or more objects that inherit their methods from a package MM. MM itself doesn't provide any methods, but it ISA ExtUtils::MM_Unix class. The inheritance tree of MM lets operating specific packages take the responsibility for all the methods provided by MM_Unix. We are trying to reduce the number of the necessary overrides by defining rather primitive operations within ExtUtils::MM_Unix.

If you are going to write a platform specific MM package, please try to limit the necessary overrides to primitive methods, and if it is not possible to do so, let's work out how to achieve that gain.

If you are overriding any of these methods in your Makefile.PL (in the MY class), please report that to the makemaker mailing list. We are trying to minimize the necessary method overrides and switch to data driven Makefile.PLs wherever possible. In the long run less methods will be overridable via the MY class.

METHODS

The following description of methods is still under development. Please refer to the code for not suitably documented sections and complain loudly to the makemaker mailing list.

Not all of the methods below are overridable in a Makefile.PL. Overridable methods are marked as (o). All methods are overridable by a platform specific MM_*.pm file (See [ExtUtils::MM_VMS](#)) and [ExtUtils::MM_OS2](#)).

SelfLoaded methods

`c_o` (o)

Defines the suffix rules to compile different flavors of C files to object files.

`cflags` (o)

Does very much the same as the cflags script in the perl distribution. It doesn't return the whole compiler command line, but initializes all of its parts. The `const_cccmd` method then actually returns the definition of the CCCMD macro which uses these parts.

`clean` (o)

Defines the clean target.

`const_cccmd` (o)

Returns the full compiler call for C programs and stores the definition in `CONST_CCCMD`.

`const_config` (o)

Defines a couple of constants in the Makefile that are imported from `%Config`.

`const_loadlibs` (o)

Defines `EXTRALIBS`, `LDLOADLIBS`, `BSLOADLIBS`, `LD_RUN_PATH`. See [ExtUtils::Liblist](#) for details.

`constants` (o)

Initializes lots of constants and `.SUFFIXES` and `.PHONY`

`depend` (o)

Same as macro for the depend attribute.

dir_target (o)

Takes an array of directories that need to exist and returns a Makefile entry for a .exists file in these directories. Returns nothing, if the entry has already been processed. We're helpless though, if the same directory comes as \$ (FOO) _and_ as "bar". Both of them get an entry, that's why we use "::<".

dist (o)

Defines a lot of macros for distribution support.

dist_basics (o)

Defines the targets distclean, distcheck, skipcheck, manifest, veryclean.

dist_ci (o)

Defines a check in target for RCS.

dist_core (o)

Defines the targets dist, tardist, zipdist, uutardist, shdist

dist_dir

Defines the scratch directory target that will hold the distribution before tar-ing (or shar-ing).

dist_test

Defines a target that produces the distribution in the scratchdirectory, and runs 'perl Makefile.PL; make ;make test' in that subdirectory.

dlsyms (o)

Used by AIX and VMS to define DL_FUNCS and DL_VARS and write the *.exp files.

dynamic (o)

Defines the dynamic target.

dynamic_bs (o)

Defines targets for bootstrap files.

dynamic_lib (o)

Defines how to produce the *.so (or equivalent) files.

exescan

Deprecated method. Use libscan instead.

extliblist

Called by init_others, and calls ext ExtUtils::Liblist. See [ExtUtils::Liblist](#) for details.

find_perl

Finds the executables PERL and FULLPERL

find_tests

```
my $test = $mm->find_tests;
```

Returns a string suitable for feeding to the shell to return all tests in t/*.t.

Methods to actually produce chunks of text for the Makefile

The methods here are called for each MakeMaker object in the order specified by @ExtUtils::MakeMaker::MM_Sections.

fixin

```
$mm->fixin(@files);
```

Inserts the sharpbang or equivalent magic number to a set of @files.

force (o)

Just writes FORCE:

guess_name

Guess the name of this package by examining the working directory's name. MakeMaker calls this only if the developer has not supplied a NAME attribute.

has_link_code

Returns true if C, XS, MYEXTLIB or similar objects exist within this object that need a compiler. Does not descend into subdirectories as `needs_linking()` does.

init_dirscan

Initializes DIR, XS, PM, C, O_FILES, H, PL_FILES, MAN*PODS, EXE_FILES.

init_main

Initializes AR, AR_STATIC_ARGS, BASEEXT, CONFIG, DISTNAME, DLBASE, EXE_EXT, FULLEXT, FULLPERL, FULLPERLRUN, FULLPERLRUNINST, INST_*, INSTALL*, INSTALLDIRS, LD, LIB_EXT, LIBPERL_A, MAP_TARGET, NAME, OBJ_EXT, PARENT_NAME, PERL, PERL_ARCHLIB, PERL_INC, PERL_LIB, PERL_SRC, PERLRUN, PERLRUNINST, PREFIX, VERSION, VERSION_FROM, VERSION_SYM, XS_VERSION.

init_others

Initializes EXTRALIBS, BSLOADLIBS, LDLOADLIBS, LIBS, LD_RUN_PATH, OBJECT, BOOTDEP, PERLMAINCC, LDFROM, LINKTYPE, NOOP, FIRST_MAKEFILE, MAKEFILE, NOECHO, RM_F, RM_RF, TEST_F, TOUCH, CP, MV, CHMOD, UMASK_NULL

init_INST

```
$mm->init_INST;
```

Called by `init_main`. Sets up all `INST_*` variables.

init_INSTALL

```
$mm->init_INSTALL;
```

Called by `init_main`. Sets up all `INSTALL_*` variables (except `INSTALLDIRS`) and `PREFIX`.

init_lib2arch

```
$mm->init_lib2arch
```

```
=end _protected
```

init_PERL

```
$mm->init_PERL;
```

Called by `init_main`. Sets up `ABSPERL`, `PERL`, `FULLPERL` and all the `*PERLRUN*` permutations.

PERL is allowed to be `miniperl`

FULLPERL must be a complete perl

ABSPERL is PERL converted to an absolute path

`*PERLRUN` contains everything necessary to run perl, find it's libraries, etc...

`*PERLRUNINST` is `*PERLRUN` + everything necessary to find the modules being built.

init_PERM

```
$mm->init_PERM
```

Called by `init_main`. Initializes `PERL_*`

install (o)

Defines the install target.

installbin (o)

Defines targets to make and to install EXE_FILES.

libscan (o)

Takes a path to a file that is found by `init_dirscan` and returns false if we don't want to include this file in the library. Mainly used to exclude RCS, CVS, and SCCS directories from installation.

linkext (o)

Defines the linkext target which in turn defines the LINKTYPE.

lsdir

Takes as arguments a directory name and a regular expression. Returns all entries in the directory that match the regular expression.

macro (o)

Simple subroutine to insert the macros defined by the macro attribute into the Makefile.

makeaperl (o)

Called by `staticmake`. Defines how to write the Makefile to produce a static new perl.

By default the Makefile produced includes all the static extensions in the perl library. (Purified versions of library files, e.g., `DynaLoader_pure_p1_c0_032.a` are automatically ignored to avoid link errors.)

makefile (o)

Defines how to rewrite the Makefile.

manifypods (o)

Defines targets and routines to translate the pods into manpages and put them into the `INST_*` directories.

maybe_command

Returns true, if the argument is likely to be a command.

maybe_command_in_dirs

method under development. Not yet used. Ask Ilya :-)

needs_linking (o)

Does this module need linking? Looks into subdirectory objects (see also `has_link_code()`)

nicetext

misnamed method (will have to be changed). The MM_Unix method just returns the argument without further processing.

On VMS used to insure that colons marking targets are preceded by space – most Unix Makes don't need this, but it's necessary under VMS to distinguish the target delimiter from a colon appearing as part of a filespec.

parse_abstract

parse a file and return what you think is the ABSTRACT

parse_version

parse a file and return what you think is `$VERSION` in this file set to. It will return the string "undef" if it can't figure out what `$VERSION` is. `$VERSION` should be for all to see, so our `$VERSION` or plain `$VERSION` are okay, but my `$VERSION` is not.

pasthru (o)

Defines the string that is passed to recursive make calls in subdirectories.

perl_script

Takes one argument, a file name, and returns the file name, if the argument is likely to be a perl script. On MM_Unix this is true for any ordinary, readable file.

perldepend (o)

Defines the dependency from all *.h files that come with the perl distribution.

perm_rw (o)

Returns the attribute PERM_RW or the string 644. Used as the string that is passed to the chmod command to set the permissions for read/writeable files. MakeMaker chooses 644 because it has turned out in the past that relying on the umask provokes hard-to-track bug reports. When the return value is used by the perl function chmod, it is interpreted as an octal value.

perm_rwx (o)

Returns the attribute PERM_RWX or the string 755, i.e. the string that is passed to the chmod command to set the permissions for executable files. See also perl_rw.

pm_to_blib

Defines target that copies all files in the hash PM to their destination and autosplits them. See [ExtUtils::Install/DESCRIPTION](#)

post_constants (o)

Returns an empty string per default. Dedicated to overrides from within Makefile.PL after all constants have been defined.

post_initialize (o)

Returns an empty string per default. Used in Makefile.PLs to add some chunk of text to the Makefile after the object is initialized.

postamble (o)

Returns an empty string. Can be used in Makefile.PLs to write some text to the Makefile at the end.

ppd

Defines target that creates a PPD (Perl Package Description) file for a binary distribution.

prefixify

```
$MM->prefixify($var, $prefix, $new_prefix, $default);
```

Using either \$MM-{uc \$var} || \$Config{lc \$var}, it will attempt to replace it's \$prefix with a \$new_prefix. Should the \$prefix fail to match it sill simply set it to the \$new_prefix + \$default.

This is for heuristics which attempt to create directory structures that mirror those of the installed perl.

For example:

```
$MM->prefixify('installman1dir', '/usr', '/home/foo', 'man/man1');
```

this will attempt to remove '/usr' from the front of the \$MM-{INSTALLMAN1DIR} path (initializing it to \$Config{installman1dir} if necessary) and replace it with '/home/foo'. If this fails it will simply use '/home/foo/man/man1'.

processPL (o)

Defines targets to run *.PL files.

quote_paren

Backslashes parentheses () in command line arguments. Doesn't handle recursive Makefile \$(...) constructs, but handles simple ones.

realclean (o)

Defines the realclean target.

replace_manpage_separator

```
my $man_name = $MM->replace_manpage_separator($file_path);
```

Takes the name of a package, which may be a nested package, in the form 'Foo/Bar.pm' and replaces the slash with : : or something else safe for a man page file name. Returns the replacement.

static (o)

Defines the static target.

static_lib (o)

Defines how to produce the *.a (or equivalent) files.

staticmake (o)

Calls makeaperl.

subdir_x (o)

Helper subroutine for subdirs

subdirs (o)

Defines targets to process subdirectories.

test (o)

Defines the test targets.

test_via_harness (override)

For some reason which I forget, Unix machines like to have PERL_DL_NONLAZY set for tests.

test_via_script (override)

Again, the PERL_DL_NONLAZY thing.

tool_autosplit (o)

Defines a simple perl call that runs autosplit. May be deprecated by pm_to_blib soon.

tools_other (o)

Defines SHELL, LD, TOUCH, CP, MV, RM_F, RM_RF, CHMOD, UMASK_NULL in the Makefile. Also defines the perl programs MKPATH, WARN_IF_OLD_PACKLIST, MOD_INSTALL, DOC_INSTALL, and UNINSTALL.

tool_xsubpp (o)

Determines typemaps, xsubpp version, prototype behaviour.

top_targets (o)

Defines the targets all, subdirs, config, and O_FILES

writedoc

Obsolete, deprecated method. Not used since Version 5.21.

xs_c (o)

Defines the suffix rules to compile XS files to C.

xs_cpp (o)

Defines the suffix rules to compile XS files to C++.

xs_o (o)

Defines suffix rules to go from XS to object files directly. This is only intended for broken make implementations.

perl_archive

This is internal method that returns path to libperl.a equivalent to be linked to dynamic extensions. UNIX does not have one but other OSs might have one.

perl_archive_after

This is an internal method that returns path to a library which should be put on the linker command line *after* the external libraries to be linked to dynamic extensions. This may be needed if the linker is one-pass, and Perl includes some overrides for C RTL functions, such as `malloc()`.

export_list

This is internal method that returns name of a file that is passed to linker to define symbols to be exported. UNIX does not have one but OS2 and Win32 do.

SEE ALSO

[*ExtUtils::MakeMaker*](#)

NAME

ExtUtils::MM_UWIN – U/WIN specific subclass of ExtUtils::MM_Unix

SYNOPSIS

Don't use this module directly.
Use ExtUtils::MM and let it choose.

DESCRIPTION

This is a subclass of ExtUtils::MM_Unix which contains functionality for the AT&T U/WIN UNIX on Windows environment.

Unless otherwise stated it works just like ExtUtils::MM_Unix

Overridden methods

replace_manpage_separator

AUTHOR

Michael G Schwern <schwern@pobox.com with code from ExtUtils::MM_Unix

SEE ALSO

[ExtUtils::MM_Win32](#), [ExtUtils::MakeMaker](#)

1;

NAME

ExtUtils::MM_VMS – methods to override UN*X behaviour in ExtUtils::MakeMaker

SYNOPSIS

Do not use this directly.
Instead, use ExtUtils::MM and it will figure out which MM_*
class to use for you.

DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

Methods always loaded

wraplist

Converts a list into a string wrapped at approximately 80 columns.

Methods

Those methods which override default MM_Unix methods are marked "(override)", while methods unique to MM_VMS are marked "(specific)". For overridden methods, documentation is limited to an explanation of why this method overrides the MM_Unix method; see the ExtUtils::MM_Unix documentation for more details.

guess_name (override)

Try to determine name of extension being built. We begin with the name of the current directory. Since VMS filenames are case-insensitive, however, we look for a *.pm* file whose name matches that of the current directory (presumably the 'main' *.pm* file for this extension), and try to find a package statement from which to obtain the Mixed::Case package name.

find_perl (override)

Use VMS file specification syntax and CLI commands to find and invoke Perl images.

maybe_command (override)

Follows VMS naming conventions for executable files. If the name passed in doesn't exactly match an executable file, appends *.Exe* (or equivalent) to check for executable image, and *.Com* to check for DCL procedure. If this fails, checks directories in DCL\$PATH and finally *Sys\$System:* for an executable file having the name specified, with or without the *.Exe*-equivalent suffix.

maybe_command_in_dirs (override)

Uses DCL argument quoting on test command line.

perl_script (override)

If name passed in doesn't specify a readable file, appends *.com* or *.pl* and tries again, since it's customary to have file types on all files under VMS.

replace_manpage_separator

Use as separator a character which is legal in a VMS-syntax file name.

init_main (override)

Override DISTVNAME so it uses VERSION_SYM to avoid getting too many dots in the name.

init_others (override)

Provide VMS-specific forms of various utility commands, then hand off to the default MM_Unix method.

constants (override)

Fixes up numerous file and directory macros to insure VMS syntax regardless of input syntax. Also adds a few VMS-specific macros and makes lists of files comma-separated.

cflags (override)

Bypass shell script and produce qualifiers for CC directly (but warn user if a shell script for this extension exists). Fold multiple /Defines into one, since some C compilers pay attention to only one instance of this qualifier on the command line.

const_cccmd (override)

Adds directives to point C preprocessor to the right place when handling `#include <sys/foo.h>` directives. Also constructs CC command line a bit differently than MM_Unix method.

pm_to_blib (override)

DCL *still* accepts a maximum of 255 characters on a command line, so we write the (potentially) long list of file names to a temp file, then persuade Perl to read it instead of the command line to find args.

tool_autosplit (override)

Use VMS-style quoting on command line.

tool_ssubpp (override)

Use VMS-style quoting on xsubpp command line.

xsubpp_version (override)

Test xsubpp exit status according to VMS rules (`$sts & 1 ==> good`) rather than Unix rules (`$sts == 0 ==> good`).

tools_other (override)

Adds a few MM[SK] macros, and shortens some the installatin commands, in order to stay under DCL's 255-character limit. Also changes EQUALIZE_TIMESTAMP to set revision date of target file to one second later than source file, since MMK interprets precisely equal revision dates for a source and target file as a sign that the target needs to be updated.

dist (override)

Provide VMSish defaults for some values, then hand off to default MM_Unix method.

c_o (override)

Use VMS syntax on command line. In particular, `$(DEFINE)` and `$(PERL_INC)` have been pulled into `$(CCCMD)`. Also use MM[SK] macros.

xs_c (override)

Use MM[SK] macros.

xs_o (override)

Use MM[SK] macros, and VMS command line for C compiler.

top_targets (override)

Path separator differences.

dlsyms (override)

Create VMS linker options files specifying universal symbols for this extension's shareable image, and listing other shareable images or libraries to which it should be linked.

dynamic_lib (override)

Use VMS Link command.

dynamic_bs (override)

Use VMS-style quoting on Mkbootstrap command line.

static_lib (override)

Use VMS commands to manipulate object library.

manifypods (override)

Use VMS-style quoting on command line, and VMS logical name to specify fallback location at build time if we can't find pod2man.

processPL (override)

Use VMS-style quoting on command line.

installbin (override)

Stay under DCL's 255 character command line limit once again by splitting potentially long list of files across multiple lines in `realclean` target.

subdir_x (override)

Use VMS commands to change default directory.

clean (override)

Split potentially long list of files across multiple commands (in order to stay under the magic command line limit). Also use MM[SK] commands for handling subdirectories.

realclean (override)

Guess what we're working around? Also, use MM[SK] for subdirectories.

dist_core (override)

Syntax for invoking *VMS_Share* differs from that for Unix *shar*, so `shdist` target actions are VMS-specific.

dist_test (override)

Use VMS commands to change default directory, and use VMS-style quoting on command line.

install (override)

Work around DCL's 255 character limit several times, and use VMS-style command line quoting in a few cases.

perldepend (override)

Use VMS-style syntax for files; it's cheaper to just do it directly here than to have the MM_Unix method call `catfile` repeatedly. Also, if we have to rebuild `Config.pm`, use MM[SK] to do it.

makefile (override)

Use VMS commands and quoting.

find_tests (override)**test (override)**

Use VMS commands for handling subdirectories.

makeaperl (override)

Undertake to build a new set of Perl images using VMS commands. Since VMS does dynamic loading, it's not necessary to statically link each extension into the Perl image, so this isn't the normal build path. Consequently, it hasn't really been tested, and may well be incomplete.

nicetext (override)

Insure that colons marking targets are preceded by space, in order to distinguish the target delimiter from a colon appearing as part of a filespec.

prefixify (override)

prefixifying on VMS is simple. Each should simply be:

```
perl_root:[some.dir]
```

which can just be converted to:

volume: [your.prefix.some.dir]

otherwise you get the default layout.

In effect, your search prefix is ignored and `$Config{vms_prefix}` is used instead.

NAME

ExtUtils::MM_Win32 – methods to override UN*X behaviour in ExtUtils::MakeMaker

SYNOPSIS

```
use ExtUtils::MM_Win32; # Done internally by ExtUtils::MakeMaker if needed
```

DESCRIPTION

See ExtUtils::MM_Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

constants (o)

Initializes lots of constants and .SUFFIXES and .PHONY

static_lib (o)

Defines how to produce the *.a (or equivalent) files.

dynamic_bs (o)

Defines targets for bootstrap files.

dynamic_lib (o)

Defines how to produce the *.so (or equivalent) files.

perl_script

Takes one argument, a file name, and returns the file name, if the argument is likely to be a perl script. On MM_Unix this is true for any ordinary, readable file.

pm_to_blib

Defines target that copies all files in the hash PM to their destination and autosplits them. See [ExtUtils::Install/DESCRIPTION](#)

tool_autosplit (override)

Use Win32 quoting on command line.

tools_other (o)

Win32 overrides.

Defines SHELL, LD, TOUCH, CP, MV, RM_F, RM_RF, CHMOD, UMASK_NULL in the Makefile. Also defines the perl programs MKPATH, WARN_IF_OLD_PACKLIST, MOD_INSTALL, DOC_INSTALL, and UNINSTALL.

xs_o (o)

Defines suffix rules to go from XS to object files directly. This is only intended for broken make implementations.

top_targets (o)

Defines the targets all, subdirs, config, and O_FILES

manifypods (o)

We don't want manpage process.

dist_ci (o)

Same as MM_Unix version (changes command-line quoting).

dist_core (o)

Same as MM_Unix version (changes command-line quoting).

pasthru (o)

Defines the string that is passed to recursive make calls in subdirectories.

NAME

ExtUtils::MM_Win95 – method to customize MakeMaker for Win9X

SYNOPSIS

You should not be using this module directly.

DESCRIPTION

This is a subclass of ExtUtils::MM_Win32 containing changes necessary to get MakeMaker playing nice with command.com and other Win9Xisms.

NAME

ExtUtils::MY – ExtUtils::MakeMaker subclass for customization

SYNOPSIS

```
# in your Makefile.PL
sub MY::whatever {
    ...
}
```

DESCRIPTION**FOR INTERNAL USE ONLY**

ExtUtils::MY is a subclass of ExtUtils::MM. Its provided in your Makefile.PL for you to add and override MakeMaker functionality.

It also provides a convenient alias via the MY class.

ExtUtils::MY might turn out to be a temporary solution, but MY won't go away.

NAME

ExtUtils::Packlist – manage .packlist files

SYNOPSIS

```
use ExtUtils::Packlist;
my ($pl) = ExtUtils::Packlist->new('.packlist');
$pl->read('/an/old/.packlist');
my @missing_files = $pl->validate();
$pl->write('/a/new/.packlist');

$pl->{'/some/file/name'}++;
    or
$pl->{'/some/other/file/name'} = { type => 'file',
                                   from => '/some/file' };
```

DESCRIPTION

ExtUtils::Packlist provides a standard way to manage .packlist files. Functions are provided to read and write .packlist files. The original .packlist format is a simple list of absolute pathnames, one per line. In addition, this package supports an extended format, where as well as a filename each line may contain a list of attributes in the form of a space separated list of key=value pairs. This is used by the installperl script to differentiate between files and links, for example.

USAGE

The hash reference returned by the `new()` function can be used to examine and modify the contents of the .packlist. Items may be added/deleted from the .packlist by modifying the hash. If the value associated with a hash key is a scalar, the entry written to the .packlist by any subsequent `write()` will be a simple filename. If the value is a hash, the entry written will be the filename followed by the key=value pairs from the hash. Reading back the .packlist will recreate the original entries.

FUNCTIONS

`new()`

This takes an optional parameter, the name of a .packlist. If the file exists, it will be opened and the contents of the file will be read. The `new()` method returns a reference to a hash. This hash holds an entry for each line in the .packlist. In the case of old-style .packlists, the value associated with each key is undef. In the case of new-style .packlists, the value associated with each key is a hash containing the key=value pairs following the filename in the .packlist.

`read()`

This takes an optional parameter, the name of the .packlist to be read. If no file is specified, the .packlist specified to `new()` will be read. If the .packlist does not exist, `Carp::croak` will be called.

`write()`

This takes an optional parameter, the name of the .packlist to be written. If no file is specified, the .packlist specified to `new()` will be overwritten.

`validate()`

This checks that every file listed in the .packlist actually exists. If an argument which evaluates to true is given, any missing files will be removed from the internal hash. The return value is a list of the missing files, which will be empty if they all exist.

`packlist_file()`

This returns the name of the associated .packlist file

EXAMPLE

Here's `modrm`, a little utility to cleanly remove an installed module.

```
#!/usr/local/bin/perl -w
```

```

use strict;
use IO::Dir;
use ExtUtils::Packlist;
use ExtUtils::Installed;

sub emptydir($) {
    my ($dir) = @_ ;
    my $dh = IO::Dir->new($dir) || return(0);
    my @count = $dh->read();
    $dh->close();
    return(@count == 2 ? 1 : 0);
}

# Find all the installed packages
print("Finding all installed modules...\n");
my $installed = ExtUtils::Installed->new();

foreach my $module (grep(!/^Perl$/, $installed->modules())) {
    my $version = $installed->version($module) || "???" ;
    print("Found module $module Version $version\n");
    print("Do you want to delete $module? [n] ");
    my $r = <STDIN>; chomp($r);
    if ($r && $r =~ /^y/i) {
        # Remove all the files
        foreach my $file (sort($installed->files($module))) {
            print("rm $file\n");
            unlink($file);
        }
        my $pf = $installed->packlist($module)->packlist_file();
        print("rm $pf\n");
        unlink($pf);
        foreach my $dir (sort($installed->directory_tree($module))) {
            if (emptydir($dir)) {
                print("rmdir $dir\n");
                rmdir($dir);
            }
        }
    }
}

```

AUTHOR

Alan Burlison <Alan.Burlison@uk.sun.com>

NAME

ExtUtils::testlib – add blib/* directories to @INC

SYNOPSIS

```
use ExtUtils::testlib;
```

DESCRIPTION

After an extension has been built and before it is installed it may be desirable to test it bypassing make test. By adding

```
use ExtUtils::testlib;
```

to a test program the intermediate directories used by make are added to @INC.

NAME

Fatal – replace functions with equivalents which succeed or die

SYNOPSIS

```
use Fatal qw(open close);

sub juggle { . . . }

import Fatal 'juggle';
```

DESCRIPTION

Fatal provides a way to conveniently replace functions which normally return a false value when they fail with equivalents which raise exceptions if they are not successful. This lets you use these functions without having to test their return values explicitly on each call. Exceptions can be caught using `eval{}`. See [perlfunc](#) and [perlvar](#) for details.

The do-or-die equivalents are set up simply by calling Fatal's `import` routine, passing it the names of the functions to be replaced. You may wrap both user-defined functions and overridable CORE operators (except `exec`, `system` which cannot be expressed via prototypes) in this way.

If the symbol `:void` appears in the import list, then functions named later in that import list raise an exception only when these are called in void context—that is, when their return values are ignored. For example

```
use Fatal qw/:void open close/;

# properly checked, so no exception raised on error
if(open(FH, "< /bogotic") {
    warn "bogo file, dude: $!";
}

# not checked, so error raises an exception
close FH;
```

AUTHOR

Lionel.Cons@cern.ch

prototype updates by Ilya Zakharevich ilya@math.ohio-state.edu

NAME

Fcntl – load the C Fcntl.h defines

SYNOPSIS

```
use Fcntl;  
use Fcntl qw(:DEFAULT :flock);
```

DESCRIPTION

This module is just a translation of the C *fcntl.h* file. Unlike the old mechanism of requiring a translated *fcntl.ph* file, this uses the **h2xs** program (see the Perl source distribution) and your native C compiler. This means that it has a far more likely chance of getting the numbers right.

NOTE

Only `#define` symbols get translated; you must still correctly pack up your own arguments to pass as args for locking functions, etc.

EXPORTED SYMBOLS

By default your system's `F_*` and `O_*` constants (eg, `F_DUPFD` and `O_CREAT`) and the `FD_CLOEXEC` constant are exported into your namespace.

You can request that the `flock()` constants (`LOCK_SH`, `LOCK_EX`, `LOCK_NB` and `LOCK_UN`) be provided by using the tag `:flock`. See [Exporter](#).

You can request that the old constants (`FAPPEND`, `FASYNC`, `FCREAT`, `FDEFER`, `FEXCL`, `FNDELAY`, `FNONBLOCK`, `FSYNC`, `FTRUNC`) be provided for compatibility reasons by using the tag `:Fcompat`. For new applications the newer versions of these constants are suggested (`O_APPEND`, `O_ASYNC`, `O_CREAT`, `O_DEFER`, `O_EXCL`, `O_NDELAY`, `O_NONBLOCK`, `O_SYNC`, `O_TRUNC`).

For ease of use also the `SEEK_*` constants (for `seek()` and `sysseek()`, e.g. `SEEK_END`) and the `S_I*` constants (for `chmod()` and `stat()`) are available for import. They can be imported either separately or using the tags `:seek` and `:mode`.

Please refer to your native `fcntl(2)`, `open(2)`, `fseek(3)`, `lseek(2)` (equal to Perl's `seek()` and `sysseek()`, respectively), and `chmod(2)` documentation to see what constants are implemented in your system.

See [perlopentut](#) to learn about the uses of the `O_*` constants with `sysopen()`.

See [seek](#) and [sysseek](#) about the `SEEK_*` constants.

See [stat](#) about the `S_I*` constants.

NAME

fields – compile-time class fields

SYNOPSIS

```
{
    package Foo;
    use fields qw(foo bar _Foo_private);
    sub new {
        my Foo $self = shift;
        unless (ref $self) {
            $self = fields::new($self);
            $self->{_Foo_private} = "this is Foo's secret";
        }
        $self->{foo} = 10;
        $self->{bar} = 20;
        return $self;
    }
}

my Foo $var = Foo::->new;
$var->{foo} = 42;

# this will generate a compile-time error
$var->{zap} = 42;

# subclassing
{
    package Bar;
    use base 'Foo';
    use fields qw(baz _Bar_private);          # not shared with Foo
    sub new {
        my $class = shift;
        my $self = fields::new($class);
        $self->SUPER::new();                  # init base fields
        $self->{baz} = 10;                    # init own fields
        $self->{_Bar_private} = "this is Bar's secret";
        return $self;
    }
}
```

DESCRIPTION

The `fields` pragma enables compile-time verified class fields.

NOTE: The current implementation keeps the declared fields in the `%FIELDS` hash of the calling package, but this may change in future versions. Do **not** update the `%FIELDS` hash directly, because it must be created at compile-time for it to be fully useful, as is done by this pragma.

If a typed lexical variable holding a reference is used to access a hash element and a package with the same name as the type has declared class fields using this pragma, then the operation is turned into an array access at compile time.

The related `base` pragma will combine fields from base classes and any fields declared using the `fields` pragma. This enables field inheritance to work properly.

Field names that start with an underscore character are made private to the class and are not visible to subclasses. Inherited fields can be overridden but will generate a warning if used together with the `-w` switch.

The effect of all this is that you can have objects with named fields which are as compact and as fast arrays to access. This only works as long as the objects are accessed through properly typed variables. If the objects are not typed, access is only checked at run time.

The following functions are supported:

new `fields::new()` creates and blesses a pseudo-hash comprised of the fields declared using the `fields` pragma into the specified class. This makes it possible to write a constructor like this:

```
package Critter::Sounds;
use fields qw(cat dog bird);

sub new {
    my Critter::Sounds $self = shift;
    $self = fields::new($self) unless ref $self;
    $self->{cat} = 'meow';                # scalar element
    @$self{'dog','bird'} = ('bark','tweet'); # slice
    return $self;
}
```

phash `fields::phash()` can be used to create and initialize a plain (unblessed) pseudo-hash. This function should always be used instead of creating pseudo-hashes directly.

If the first argument is a reference to an array, the pseudo-hash will be created with keys from that array. If a second argument is supplied, it must also be a reference to an array whose elements will be used as the values. If the second array contains less elements than the first, the trailing elements of the pseudo-hash will not be initialized. This makes it particularly useful for creating a pseudo-hash from subroutine arguments:

```
sub dogtag {
    my $tag = fields::phash([qw(name rank ser_num)], [ @_ ]);
}
```

`fields::phash()` also accepts a list of key-value pairs that will be used to construct the pseudo hash. Examples:

```
my $tag = fields::phash(name => "Joe",
                        rank => "captain",
                        ser_num => 42);

my $pseudohash = fields::phash(%args);
```

SEE ALSO

base, *Pseudo-hashes: Using an array as a hash*

NAME

`fileparse` – split a pathname into pieces

`basename` – extract just the filename from a path

`dirname` – extract just the directory from a path

SYNOPSIS

```
use File::Basename;

($name,$path,$suffix) = fileparse($fullname,@suffixlist);
fileparse_set_fstype($os_string);
$basename = basename($fullname,@suffixlist);
$dirname = dirname($fullname);

($name,$path,$suffix) = fileparse("lib/File/Basename.pm",qr{\.pm});
fileparse_set_fstype("VMS");
$basename = basename("lib/File/Basename.pm",qr{\.pm});
$dirname = dirname("lib/File/Basename.pm");
```

DESCRIPTION

These routines allow you to parse file specifications into useful pieces using the syntax of different operating systems.

`fileparse_set_fstype`

You select the syntax via the routine `fileparse_set_fstype()`.

If the argument passed to it contains one of the substrings "VMS", "MSDOS", "MacOS", "AmigaOS" or "MSWin32", the file specification syntax of that operating system is used in future calls to `fileparse()`, `basename()`, and `dirname()`. If it contains none of these substrings, Unix syntax is used. This pattern matching is case-insensitive. If you've selected VMS syntax, and the file specification you pass to one of these routines contains a "/", they assume you are using Unix emulation and apply the Unix syntax rules instead, for that function call only.

If the argument passed to it contains one of the substrings "VMS", "MSDOS", "MacOS", "AmigaOS", "os2", "MSWin32" or "RISCOS", then the pattern matching for suffix removal is performed without regard for case, since those systems are not case-sensitive when opening existing files (though some of them preserve case on file creation).

If you haven't called `fileparse_set_fstype()`, the syntax is chosen by examining the builtin variable `$^O` according to these rules.

`fileparse`

The `fileparse()` routine divides a file specification into three parts: a leading **path**, a file **name**, and a **suffix**. The **path** contains everything up to and including the last directory separator in the input file specification. The remainder of the input file specification is then divided into **name** and **suffix** based on the optional patterns you specify in `@suffixlist`. Each element of this list can be a `qr`-quoted pattern (or a string which is interpreted as a regular expression), and is matched against the end of **name**. If this succeeds, the matching portion of **name** is removed and prepended to **suffix**. By proper use of `@suffixlist`, you can remove file types or versions for examination.

You are guaranteed that if you concatenate **path**, **name**, and **suffix** together in that order, the result will denote the same file as the input file specification.

EXAMPLES

Using Unix file syntax:

```
($base,$path,$type) = fileparse('/virgil/aeneid/draft.book7',
                                qr{\.book\d+});
```


would yield

```
$base eq 'draft'
$path eq '/virgil/aeneid/',
$type eq '.book7'
```

Similarly, using VMS syntax:

```
($name,$dir,$type) = fileparse('Doc_Root:[Help]Rhetoric.Rnh',
                                qr{\.*});
```

would yield

```
$name eq 'Rhetoric'
$dir  eq 'Doc_Root:[Help]'
$type eq '.Rnh'
```

basename

The `basename()` routine returns the first element of the list produced by calling `fileparse()` with the same arguments, except that it always quotes metacharacters in the given suffixes. It is provided for programmer compatibility with the Unix shell command `basename(1)`.

dirname

The `dirname()` routine returns the directory portion of the input file specification. When using VMS or MacOS syntax, this is identical to the second element of the list produced by calling `fileparse()` with the same input file specification. (Under VMS, if there is no directory information in the input file specification, then the current default device and directory are returned.) When using Unix or MSDOS syntax, the return value conforms to the behavior of the Unix shell command `dirname(1)`. This is usually the same as the behavior of `fileparse()`, but differs in some cases. For example, for the input file specification *lib/*, `fileparse()` considers the directory name to be *lib/*, while `dirname()` considers the directory name to be *.*).

NAME

validate – run many filetest checks on a tree

SYNOPSIS

```
use File::CheckTree;

$num_warnings = validate( q{
    /vmunix          -e || die
    /boot             -e || die
    /bin              cd
        csh           -ex
        csh           !-ug
        sh            -ex
        sh            !-ug
    /usr              -d || warn "What happened to $file?\n"
});
```

DESCRIPTION

The `validate()` routine takes a single multiline string consisting of directives, each containing a filename plus a file test to try on it. (The file test may also be a "cd", causing subsequent relative filenames to be interpreted relative to that directory.) After the file test you may put `|| die` to make it a fatal error if the file test fails. The default is `|| warn`. The file test may optionally have a "!" prepended to test for the opposite condition. If you do a `cd` and then list some relative filenames, you may want to indent them slightly for readability. If you supply your own `die()` or `warn()` message, you can use `$file` to interpolate the filename.

Filetests may be bunched: `"-rwx"` tests for all of `-r`, `-w`, and `-x`. Only the first failed test of the bunch will produce a warning.

The routine returns the number of warnings issued.

AUTHOR

Unknown. Revised by Paul Grassie <grassie@perl.com> in 2002.

HISTORY

`File::CheckTree` used to not display fatal error messages. It used to count only those warnings produced by a generic `|| warn` (and not those in which the user supplied the message). In addition, the `validate()` routine would leave the user program in whatever directory was last entered through the use of "cd" directives. These bugs were fixed during the development of perl 5.8. The first fixed version of `File::CheckTree` was 4.2.

NAME

File::Compare – Compare files or filehandles

SYNOPSIS

```
use File::Compare;

if (compare("file1","file2") == 0) {
    print "They're equal\n";
}
```

DESCRIPTION

The File::Compare::compare function compares the contents of two sources, each of which can be a file or a file handle. It is exported from File::Compare by default.

File::Compare::cmp is a synonym for File::Compare::compare. It is exported from File::Compare only by request.

File::Compare::compare_text does a line by line comparison of the two files. It stops as soon as a difference is detected. compare_text() accepts an optional third argument: This must be a CODE reference to a line comparison function, which returns 0 when both lines are considered equal. For example:

```
compare_text($file1, $file2)
```

is basically equivalent to

```
compare_text($file1, $file2, sub {$_[0] ne $_[1]} )
```

RETURN

File::Compare::compare and its sibling functions return 0 if the files are equal, 1 if the files are unequal, or -1 if an error was encountered.

AUTHOR

File::Compare was written by Nick Ing-Simmons. Its original documentation was written by Chip Salzenberg.

NAME

File::Copy – Copy files or filehandles

SYNOPSIS

```
use File::Copy;

copy("file1", "file2");
copy("Copy.pm", \*STDOUT);
move("/dev1/fileA", "/dev2/fileB");

use POSIX;
use File::Copy cp;

$n = FileHandle->new("/a/file", "r");
cp($n, "x");
```

DESCRIPTION

The File::Copy module provides two basic functions, `copy` and `move`, which are useful for getting the contents of a file from one place to another.

- The `copy` function takes two parameters: a file to copy from and a file to copy to. Either argument may be a string, a FileHandle reference or a FileHandle glob. Obviously, if the first argument is a filehandle of some sort, it will be read from, and if it is a file *name* it will be opened for reading. Likewise, the second argument will be written to (and created if need be). Trying to copy a file on top of itself is a fatal error.

Note that passing in files as handles instead of names may lead to loss of information on some operating systems; it is recommended that you use file names whenever possible. Files are opened in binary mode where applicable. To get a consistent behaviour when copying from a filehandle to a file, use `binmode` on the filehandle.

An optional third parameter can be used to specify the buffer size used for copying. This is the number of bytes from the first file, that will be held in memory at any given time, before being written to the second file. The default buffer size depends upon the file, but will generally be the whole file (up to 2Mb), or 1k for filehandles that do not reference files (eg. sockets).

You may use the syntax `use File::Copy "cp"` to get at the "cp" alias for this function. The syntax is *exactly* the same.

- The `move` function also takes two parameters: the current name and the intended name of the file to be moved. If the destination already exists and is a directory, and the source is not a directory, then the source file will be renamed into the directory specified by the destination.

If possible, `move()` will simply rename the file. Otherwise, it copies the file to the new location and deletes the original. If an error occurs during this copy-and-delete process, you may be left with a (possibly partial) copy of the file under the destination name.

You may use the "mv" alias for this function in the same way that you may use the "cp" alias for `copy`.

File::Copy also provides the `syscopy` routine, which copies the file specified in the first parameter to the file specified in the second parameter, preserving OS-specific attributes and file structure. For Unix systems, this is equivalent to the simple `copy` routine, which doesn't preserve OS-specific attributes. For VMS systems, this calls the `rmscopy` routine (see below). For OS/2 systems, this calls the `syscopy` XSUB directly. For Win32 systems, this calls `Win32::CopyFile`.

On Mac OS (Classic), `syscopy` calls `Mac::MoreFiles::FSpFileCopy`, if available.

Special behaviour if `syscopy` is defined (OS/2, VMS and Win32)

If both arguments to `copy` are not file handles, then `copy` will perform a "system copy" of the input file to a new output file, in order to preserve file attributes, indexed file structure, *etc.* The buffer size parameter is ignored. If either argument to `copy` is a handle to an opened file, then data is copied using Perl operators, and no effort is made to preserve file attributes or record structure.

The system copy routine may also be called directly under VMS and OS/2 as `File::Copy::syscopy` (or under VMS as `File::Copy::rmscopy`, which is the routine that does the actual work for `syscopy`).

`rmscopy($from, $to[, $date_flag])`

The first and second arguments may be strings, typeglobs, typeglob references, or objects inheriting from `IO::Handle`; they are used in all cases to obtain the *filespec* of the input and output files, respectively. The name and type of the input file are used as defaults for the output file, if necessary.

A new version of the output file is always created, which inherits the structure and RMS attributes of the input file, except for owner and protections (and possibly timestamps; see below). All data from the input file is copied to the output file; if either of the first two parameters to `rmscopy` is a file handle, its position is unchanged. (Note that this means a file handle pointing to the output file will be associated with an old version of that file after `rmscopy` returns, not the newly created version.)

The third parameter is an integer flag, which tells `rmscopy` how to handle timestamps. If it is `< 0`, none of the input file's timestamps are propagated to the output file. If it is `> 0`, then it is interpreted as a bitmask: if bit 0 (the LSB) is set, then timestamps other than the revision date are propagated; if bit 1 is set, the revision date is propagated. If the third parameter to `rmscopy` is 0, then it behaves much like the DCL COPY command: if the name or type of the output file was explicitly specified, then no timestamps are propagated, but if they were taken implicitly from the input filespec, then all timestamps other than the revision date are propagated. If this parameter is not supplied, it defaults to 0.

Like `copy`, `rmscopy` returns 1 on success. If an error occurs, it sets `$!`, deletes the output file, and returns 0.

RETURN

All functions return 1 on success, 0 on failure. `$!` will be set if an error was encountered.

NOTES

- On Mac OS (Classic), the path separator is `':'`, not `'/'`, and the current directory is denoted as `':'`, not `'.'`. You should be careful about specifying relative pathnames. While a full path always begins with a volume name, a relative pathname should always begin with a `':'`. If specifying a volume name only, a trailing `':'` is required.

E.g.

```
copy("file1", "tmp");           # creates the file 'tmp' in the current directory
copy("file1", ":tmp:");         # creates :tmp:file1
copy("file1", ":tmp");          # same as above
copy("file1", "tmp");           # same as above, if 'tmp' is a directory (but don't
                                # that, since it may cause confusion, see example
copy("file1", "tmp:file1");     # error, since 'tmp:' is not a volume
copy("file1", ":tmp:file1");    # ok, partial path
copy("file1", "DataHD:");       # creates DataHD:file1

move("MacintoshHD:fileA", "DataHD:fileB"); # moves (don't copies) files from on
                                           # volume to another
```

AUTHOR

File::Copy was written by Aaron Sherman <*ajs@ajs.com*> in 1995, and updated by Charles Bailey <*bailey@newman.upenn.edu*> in 1996.

NAME

File::DosGlob – DOS like globbing and then some

SYNOPSIS

```
require 5.004;

# override CORE::glob in current package
use File::DosGlob 'glob';

# override CORE::glob in ALL packages (use with extreme caution!)
use File::DosGlob 'GLOBAL_glob';

@perlfiles = glob "..\\pe?l/*.p?";
print <..\\pe?l/*.p?>;

# from the command line (overrides only in main::)
> perl -MFile::DosGlob=glob -e "print <../pe*/*p?>"
```

DESCRIPTION

A module that implements DOS-like globbing with a few enhancements. It is largely compatible with `perlglob.exe` (the M\$ `setargv.obj` version) in all but one respect—it understands wildcards in directory components.

For example, `<..\\l*b\\file/*glob.p?>` will work as expected (in that it will find something like `'..\\lib\\File\\DosGlob.pm'` alright). Note that all path components are case-insensitive, and that backslashes and forward slashes are both accepted, and preserved. You may have to double the backslashes if you are putting them in literally, due to double-quotish parsing of the pattern by perl.

Spaces in the argument delimit distinct patterns, so `glob('*.*exe *.dll')` globs all filenames that end in `.exe` or `.dll`. If you want to put in literal spaces in the glob pattern, you can escape them with either double quotes, or backslashes. e.g. `glob('c:/"Program Files"/*/*.dll')`, or `glob('c:/Program\ Files/*/*.dll')`. The argument is tokenized using `Text::ParseWords::parse_line()`, so see [Text::ParseWords](#) for details of the quoting rules used.

Extending it to csh patterns is left as an exercise to the reader.

NOTES

- Mac OS (Classic) users should note a few differences. The specification of pathnames in glob patterns adheres to the usual Mac OS conventions: The path separator is a colon `:`, not a slash `/` or backslash `\`. A full path always begins with a volume name. A relative pathname on Mac OS must always begin with a `:`, except when specifying a file or directory name in the current working directory, where the leading colon is optional. If specifying a volume name only, a trailing `:` is required. Due to these rules, a glob like `<*>` will find all mounted volumes, while a glob like `<*>` or `<:*>` will find all files and directories in the current directory.

Note that updirs in the glob pattern are resolved before the matching begins, i.e. a pattern like `"*HD:t?p::a"` will be matched as `"*HD:a"`. Note also, that a single trailing `:` in the pattern is ignored (unless it's a volume name pattern like `"*HD:"`), i.e. a glob like `<:*>` will find both directories and files (and not, as one might expect, only directories).

The metachars `*`, `?` and the escape char `\` are valid characters in volume, directory and file names on Mac OS. Hence, if you want to match a `*`, `?` or `\` literally, you have to escape these characters. Due to perl's quoting rules, things may get a bit complicated, when you want to match a string like `*` literally, or when you want to match `\` literally, but treat the immediately following character `*` as metachar. So, here's a rule of thumb (applies to both single- and double-quoted strings): escape each `*` or `?` or `\` with a backslash, if you want to treat them literally, and then double each backslash and you are done. E.g.

– Match `*` literally

```
escape both '\ ' and '*' : '\\\\*'
double the backslashes   : '\\\\\\\\*'

```

(Internally, the glob routine sees a '*', which means that both '\ ' and '*' are escaped.)

– Match '\ ' literally, treat '*' as metachar

```
escape '\ ' but not '*'   : '\\\\*'
double the backslashes    : '\\\\\\\\*'

```

(Internally, the glob routine sees a '*', which means that '\ ' is escaped and '*' is not.)

Note that you also have to quote literal spaces in the glob pattern, as described above.

EXPORTS (by request only)

`glob()`

BUGS

Should probably be built into the core, and needs to stop pandering to DOS habits. Needs a dose of optimizium too.

AUTHOR

Gurusamy Sarathy <gsar@activestate.com>

HISTORY

- Support for globally overriding `glob()` (GSAR 3–JUN–98)
- Scalar context, independent iterator context fixes (GSAR 15–SEP–97)
- A few dir-vs-file optimizations result in glob importation being 10 times faster than using `perlglob.exe`, and using `perlglob.bat` is only twice as slow as `perlglob.exe` (GSAR 28–MAY–97)
- Several cleanups prompted by lack of compatible `perlglob.exe` under Borland (GSAR 27–MAY–97)
- Initial version (GSAR 20–FEB–97)

SEE ALSO

`perl`

`perlglob.bat`

`Text::ParseWords`

NAME

File::Find – Traverse a directory tree.

SYNOPSIS

```
use File::Find;
find(\&wanted, @directories_to_search);
sub wanted { ... }

use File::Find;
finddepth(\&wanted, @directories_to_search);
sub wanted { ... }

use File::Find;
find({ wanted => \&process, follow => 1 }, '.');
```

DESCRIPTION

These are functions for searching through directory trees doing work on each file found similar to the Unix *find* command. File::Find exports two functions, *find* and *finddepth*. They work similarly but have subtle differences.

find

```
find(\&wanted, @directories);
find(\%options, @directories);
```

find() does a breadth-first search over the given @directories in the order they are given. In essence, it works from the top down.

For each file or directory found the *&wanted* subroutine is called (see below for details). Additionally, for each directory found it will go into that directory and continue the search.

finddepth

```
finddepth(\&wanted, @directories);
finddepth(\%options, @directories);
```

finddepth() works just like *find()* except it does a depth-first search. It works from the bottom of the directory tree up.

%options

The first argument to *find()* is either a hash reference describing the operations to be performed for each file, or a code reference. The code reference is described in [The wanted function](#) below.

Here are the possible keys for the hash:

wanted

The value should be a code reference. This code reference is described in [The wanted function](#) below.

bydepth

Reports the name of a directory only AFTER all its entries have been reported. Entry point *finddepth()* is a shortcut for specifying { *bydepth* => 1 } in the first argument of *find()*.

preprocess

The value should be a code reference. This code reference is used to preprocess the current directory. The name of currently processed directory is in `$File::Find::dir`. Your preprocessing function is called after *readdir()* but before the loop that calls the *wanted()* function. It is called with a list of strings (actually file/directory names) and is expected to return a list of strings. The code can be used to sort the file/directory names alphabetically, numerically, or to filter out directory entries based on their name alone. When *follow* or *follow_fast* are in effect, *preprocess* is a no-op.

postprocess

The value should be a code reference. It is invoked just before leaving the currently processed directory. It is called in void context with no arguments. The name of the current directory is in `$File::Find::dir`. This hook is handy for summarizing a directory, such as calculating its disk usage. When *follow* or *follow_fast* are in effect, *postprocess* is a no-op.

follow

Causes symbolic links to be followed. Since directory trees with symbolic links (followed) may contain files more than once and may even have cycles, a hash has to be built up with an entry for each file. This might be expensive both in space and time for a large directory tree. See *follow_fast* and *follow_skip* below. If either *follow* or *follow_fast* is in effect:

- It is guaranteed that an *lstat* has been called before the user's *wanted()* function is called. This enables fast file checks involving `_`.
- There is a variable `$File::Find::fullname` which holds the absolute pathname of the file with all symbolic links resolved

follow_fast

This is similar to *follow* except that it may report some files more than once. It does detect cycles, however. Since only symbolic links have to be hashed, this is much cheaper both in space and time. If processing a file more than once (by the user's *wanted()* function) is worse than just taking time, the option *follow* should be used.

follow_skip

`follow_skip=1`, which is the default, causes all files which are neither directories nor symbolic links to be ignored if they are about to be processed a second time. If a directory or a symbolic link are about to be processed a second time, `File::Find` dies. `follow_skip==0` causes `File::Find` to die if any file is about to be processed a second time. `follow_skip==2` causes `File::Find` to ignore any duplicate files and directories but to proceed normally otherwise.

dangling_symlinks

If true and a code reference, will be called with the symbolic link name and the directory it lives in as arguments. Otherwise, if true and warnings are on, warning "symbolic_link_name is a dangling symbolic link\n" will be issued. If false, the dangling symbolic link will be silently ignored.

no_chdir

Does not `chdir()` to each directory as it recurses. The *wanted()* function will need to be aware of this, of course. In this case, `$_` will be the same as `$File::Find::name`.

untaint

If *find* is used in taint-mode (`-T` command line switch or if `EUID != UID` or if `EGID != GID`) then internally directory names have to be untainted before they can be `chdir`'ed to. Therefore they are checked against a regular expression *untaint_pattern*. Note that all names passed to the user's *wanted()* function are still tainted. If this option is used while not in taint-mode, *untaint* is a no-op.

untaint_pattern

See above. This should be set using the `qr` quoting operator. The default is set to `qr|^([-+@\\w. /] +) $|`. Note that the parentheses are vital.

untaint_skip

If set, a directory which fails the *untaint_pattern* is skipped, including all its sub-directories. The default is to 'die' in such a case.

The wanted function

The `wanted()` function does whatever verifications you want on each file and directory. It takes no arguments but rather does its work through a collection of variables.

`$File::Find::dir` is the current directory name,

`$_` is the current filename within that directory

`$File::Find::name` is the complete pathname to the file.

Don't modify these variables.

For example, when examining the file `/some/path/foo.ext` you will have:

```
$File::Find::dir = /some/path/
$_              = foo.ext
$File::Find::name = /some/path/foo.ext
```

You are `chdir()`d to `$File::Find::dir` when the function is called, unless `no_chdir` was specified. Note that when changing to directories is in effect the root directory (`/`) is a somewhat special case inasmuch as the concatenation of `$File::Find::dir`, `'/'` and `$_` is not literally equal to `$File::Find::name`. The table below summarizes all variants:

	<code>\$File::Find::name</code>	<code>\$File::Find::dir</code>	<code>\$_</code>
default	<code>/</code>	<code>/</code>	<code>.</code>
<code>no_chdir=>0</code>	<code>/etc</code> <code>/etc/x</code>	<code>/</code> <code>/etc</code>	<code>etc</code> <code>x</code>
<code>no_chdir=>1</code>	<code>/</code> <code>/etc</code> <code>/etc/x</code>	<code>/</code> <code>/</code> <code>/etc</code>	<code>/</code> <code>/etc</code> <code>/etc/x</code>

When `<follow` or `<follow_fast` are in effect, there is also a `$File::Find::fullname`. The function may set `$File::Find::prune` to prune the tree unless `bydepth` was specified. Unless `follow` or `follow_fast` is specified, for compatibility reasons (`find.pl`, `find2perl`) there are in addition the following globals available: `$File::Find::topdir`, `$File::Find::topdev`, `$File::Find::topino`, `$File::Find::topmode` and `$File::Find::topnlink`.

This library is useful for the `find2perl` tool, which when fed,

```
find2perl / -name .nfs\* -mtime +7 \
    -exec rm -f {} \; -o -fstype nfs -prune
```

produces something like:

```
sub wanted {
    /^\.nfs.*\z/s &&
    (($dev, $ino, $mode, $nlink, $uid, $gid) = lstat($_)) &&
    int(-M _) > 7 &&
    unlink($_)
    ||
    ($nlink || (($dev, $ino, $mode, $nlink, $uid, $gid) = lstat($_)) &&
    $dev < 0 &&
    ($File::Find::prune = 1);
}
```

Notice the `_` in the above `int(-M _)`: the `_` is a magical filehandle that caches the information from the preceding `stat()`, `lstat()`, or `filetest`.

Here's another interesting wanted function. It will find all symbolic links that don't resolve:

```
sub wanted {
    -l && !-e && print "bogus link: $File::Find::name\n";
}
```

```
}
```

See also the script `pfind` on CPAN for a nice application of this module.

WARNINGS

If you run your program with the `-w` switch, or if you use the `warnings` pragma, `File::Find` will report warnings for several weird situations. You can disable these warnings by putting the statement

```
no warnings 'File::Find';
```

in the appropriate scope. See [perllexwarn](#) for more info about lexical warnings.

CAVEAT

`$dont_use_nlink`

You can set the variable `$File::Find::dont_use_nlink` to 1, if you want to force `File::Find` to always stat directories. This was used for file systems that do not have an `nlink` count matching the number of sub-directories. Examples are ISO-9660 (CD-ROM), AFS, HPFS (OS/2 file system), FAT (DOS file system) and a couple of others.

You shouldn't need to set this variable, since `File::Find` should now detect such file systems on-the-fly and switch itself to using `stat`. This works even for parts of your file system, like a mounted CD-ROM.

If you do set `$File::Find::dont_use_nlink` to 1, you will notice slow-downs.

symlinks

Be aware that the option to follow symbolic links can be dangerous. Depending on the structure of the directory tree (including symbolic links to directories) you might traverse a given (physical) directory more than once (only if `follow_fast` is in effect). Furthermore, deleting or changing files in a symbolically linked directory might cause very unpleasant surprises, since you delete or change files in an unknown directory.

NOTES

- Mac OS (Classic) users should note a few differences:
 - The path separator is `':'`, not `'/'`, and the current directory is denoted as `':'`, not `'.'`. You should be careful about specifying relative pathnames. While a full path always begins with a volume name, a relative pathname should always begin with a `':'`. If specifying a volume name only, a trailing `':'` is required.
 - `$File::Find::dir` is guaranteed to end with a `':'`. If `$_` contains the name of a directory, that name may or may not end with a `':'`. Likewise, `$File::Find::name`, which contains the complete pathname to that directory, and `$File::Find::fullname`, which holds the absolute pathname of that directory with all symbolic links resolved, may or may not end with a `':'`.
 - The default `untaint_pattern` (see above) on Mac OS is set to `qr|^(.+)$|`. Note that the parentheses are vital.
 - The invisible system file "Icon\015" is ignored. While this file may appear in every directory, there are some more invisible system files on every volume, which are all located at the volume root level (i.e. "MacintoshHD:"). These system files are **not** excluded automatically. Your filter may use the following code to recognize invisible files or directories (requires `Mac::Files`):

```
use Mac::Files;

# invisible() -- returns 1 if file/directory is invisible,
# 0 if it's visible or undef if an error occurred

sub invisible($) {
    my $file = shift;
    my ($fileCat, $fileInfo);
```

```
my $invisible_flag = 1 << 14;

if ( $fileCat = FSpGetCatInfo($file) ) {
    if ($fileInfo = $fileCat->ioFlFndrInfo() ) {
        return (($fileInfo->fdFlags & $invisible_flag) && 1);
    }
}
return undef;
}
```

Generally, invisible files are system files, unless an odd application decides to use invisible files for its own purposes. To distinguish such files from system files, you have to look at the **type** and **creator** file attributes. The MacPerl built-in functions `GetFileInfo(FILE)` and `SetFileInfo(CREATOR, TYPE, FILES)` offer access to these attributes (see `MacPerl.pm` for details).

Files that appear on the desktop actually reside in an (hidden) directory named "Desktop Folder" on the particular disk volume. Note that, although all desktop files appear to be on the same "virtual" desktop, each disk volume actually maintains its own "Desktop Folder" directory.

HISTORY

`File::Find` used to produce incorrect results if called recursively. During the development of perl 5.8 this bug was fixed. The first fixed version of `File::Find` was 1.01.

NAME

File::Glob – Perl extension for BSD glob routine

SYNOPSIS

```
use File::Glob ':glob';
@list = bsd_glob('*. [ch]');
$homedir = bsd_glob('~gnat', GLOB_TILDE | GLOB_ERR);
if (GLOB_ERROR) {
    # an error occurred reading $homedir
}

## override the core glob (CORE::glob() does this automatically
## by default anyway, since v5.6.0)
use File::Glob ':globally';
my @sources = <*. {c,h,y}>

## override the core glob, forcing case sensitivity
use File::Glob qw(:globally :case);
my @sources = <*. {c,h,y}>

## override the core glob forcing case insensitivity
use File::Glob qw(:globally :nocase);
my @sources = <*. {c,h,y}>
```

DESCRIPTION

File::Glob::bsd_glob() implements the FreeBSD glob(3) routine, which is a superset of the POSIX glob() (described in IEEE Std 1003.2 "POSIX.2"). bsd_glob() takes a mandatory pattern argument, and an optional flags argument, and returns a list of filenames matching the pattern, with interpretation of the pattern modified by the flags variable.

Since v5.6.0, Perl's CORE::glob() is implemented in terms of bsd_glob(). Note that they don't share the same prototype—CORE::glob() only accepts a single argument. Due to historical reasons, CORE::glob() will also split its argument on whitespace, treating it as multiple patterns, whereas bsd_glob() considers them as one pattern.

The POSIX defined flags for bsd_glob() are:

GLOB_ERR

Force bsd_glob() to return an error when it encounters a directory it cannot open or read. Ordinarily bsd_glob() continues to find matches.

GLOB_LIMIT

Make bsd_glob() return an error (GLOB_NOSPACE) when the pattern expands to a size bigger than the system constant ARG_MAX (usually found in limits.h). If your system does not define this constant, bsd_glob() uses sysconf(_SC_ARG_MAX) or _POSIX_ARG_MAX where available (in that order). You can inspect these values using the standard POSIX extension.

GLOB_MARK

Each pathname that is a directory that matches the pattern has a slash appended.

GLOB_NOCASE

By default, file names are assumed to be case sensitive; this flag makes bsd_glob() treat case differences as not significant.

GLOB_NOCHECK

If the pattern does not match any pathname, then bsd_glob() returns a list consisting of only the pattern. If GLOB_QUOTE is set, its effect is present in the pattern returned.

GLOB_NOSORT

By default, the pathnames are sorted in ascending ASCII order; this flag prevents that sorting (speeding up `bsd_glob()`).

The FreeBSD extensions to the POSIX standard are the following flags:

GLOB_BRACE

Pre-process the string to expand `{pat,pat,...}` strings like `csh(1)`. The pattern `'{'` is left unexpanded for historical reasons (and `csh(1)` does the same thing to ease typing of `find(1)` patterns).

GLOB_NOMAGIC

Same as **GLOB_NOCHECK** but it only returns the pattern if it does not contain any of the special characters `"*`, `"?"` or `"["`. **NOMAGIC** is provided to simplify implementing the historic `csh(1)` globbing behaviour and should probably not be used anywhere else.

GLOB_QUOTE

Use the backslash (`'\'`) character for quoting: every occurrence of a backslash followed by a character in the pattern is replaced by that character, avoiding any special interpretation of the character. (But see below for exceptions on DOSISH systems).

GLOB_TILDE

Expand patterns that start with `'~'` to user name home directories.

GLOB_CSH

For convenience, **GLOB_CSH** is a synonym for **GLOB_BRACE** | **GLOB_NOMAGIC** | **GLOB_QUOTE** | **GLOB_TILDE** | **GLOB_ALPHASORT**.

The POSIX provided **GLOB_APPEND**, **GLOB_DOOFFS**, and the FreeBSD extensions **GLOB_ALTDIRFUNC**, and **GLOB_MAGCHAR** flags have not been implemented in the Perl version because they involve more complex interaction with the underlying C structures.

The following flag has been added in the Perl implementation for `csh` compatibility:

GLOB_ALPHASORT

If **GLOB_NOSORT** is not in effect, sort filenames in alphabetical order (case does not matter) rather than in ASCII order.

DIAGNOSTICS

`bsd_glob()` returns a list of matching paths, possibly zero length. If an error occurred, `&File::Glob::GLOB_ERROR` will be non-zero and `$!` will be set. `&File::Glob::GLOB_ERROR` is guaranteed to be zero if no error occurred, or one of the following values otherwise:

GLOB_NOSPACE

An attempt to allocate memory failed.

GLOB_ABEND

The glob was stopped because an error was encountered.

In the case where `bsd_glob()` has found some matching paths, but is interrupted by an error, it will return a list of filenames **and** set `&File::Glob::ERROR`.

Note that `bsd_glob()` deviates from POSIX and FreeBSD `glob(3)` behaviour by not considering `ENOENT` and `ENOTDIR` as errors – `bsd_glob()` will continue processing despite those errors, unless the **GLOB_ERR** flag is set.

Be aware that all filenames returned from `File::Glob` are tainted.

NOTES

- If you want to use multiple patterns, e.g. `bsd_glob "a* b*"`, you should probably throw them in a set as in `bsd_glob "{a*,b*}"`. This is because the argument to `bsd_glob()` isn't subjected to parsing by the C shell. Remember that you can use a backslash to escape things.
- On DOSISH systems, backslash is a valid directory separator character. In this case, use of backslash as a quoting character (via `GLOB_QUOTE`) interferes with the use of backslash as a directory separator. The best (simplest, most portable) solution is to use forward slashes for directory separators, and backslashes for quoting. However, this does not match "normal practice" on these systems. As a concession to user expectation, therefore, backslashes (under `GLOB_QUOTE`) only quote the glob metacharacters `['', ']', '{', '}', '-', '~'`, and backslash itself. All other backslashes are passed through unchanged.
- Win32 users should use the real slash. If you really want to use backslashes, consider using Sarathy's `File::DosGlob`, which comes with the standard Perl distribution.
- Mac OS (Classic) users should note a few differences. Since Mac OS is not Unix, when the glob code encounters a tilde glob (e.g. `~user`) and the `GLOB_TILDE` flag is used, it simply returns that pattern without doing any expansion.

Glob on Mac OS is case-insensitive by default (if you don't use any flags). If you specify any flags at all and still want glob to be case-insensitive, you must include `GLOB_NOCASE` in the flags.

The path separator is `':'` (aka colon), not `'/'` (aka slash). Mac OS users should be careful about specifying relative pathnames. While a full path always begins with a volume name, a relative pathname should always begin with a `':'`. If specifying a volume name only, a trailing `':'` is required.

The specification of pathnames in glob patterns adheres to the usual Mac OS conventions: The path separator is a colon `':'`, not a slash `'/'`. A full path always begins with a volume name. A relative pathname on Mac OS must always begin with a `':'`, except when specifying a file or directory name in the current working directory, where the leading colon is optional. If specifying a volume name only, a trailing `':'` is required. Due to these rules, a glob like `<*:>` will find all mounted volumes, while a glob like `<*>` or `<:*>` will find all files and directories in the current directory.

Note that updirs in the glob pattern are resolved before the matching begins, i.e. a pattern like `"*HD:t?p::a*"` will be matched as `"*HD:a*"`. Note also, that a single trailing `':'` in the pattern is ignored (unless it's a volume name pattern like `"*HD:"`), i.e. a glob like `<*:*>` will find both directories *and* files (and not, as one might expect, only directories). You can, however, use the `GLOB_MARK` flag to distinguish (without a file test) directory names from file names.

If the `GLOB_MARK` flag is set, all directory paths will have a `':'` appended. Since a directory like `'lib:'` is *not* a valid *relative* path on Mac OS, both a leading and a trailing colon will be added, when the directory name in question doesn't contain any colons (e.g. `'lib'` becomes `':lib:'`).

AUTHOR

The Perl interface was written by Nathan Torkington <gnat@frii.com>, and is released under the artistic license. Further modifications were made by Greg Bacon <gbacon@cs.uah.edu>, Gurusamy Sarathy <gsar@activestate.com>, and Thomas Wegner <wegner_thomas@yahoo.com>. The C glob code has the following copyright:

Copyright (c) 1989, 1993 The Regents of the University of California.
All rights reserved.

This code is derived from software contributed to Berkeley by
Guido van Rossum.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

NAME

File::Path – create or remove directory trees

SYNOPSIS

```
use File::Path;

mkpath(['foo/bar/baz', 'blurfl/quux'], 1, 0711);
rmtree(['foo/bar/baz', 'blurfl/quux'], 1, 1);
```

DESCRIPTION

The `mkpath` function provides a convenient way to create directories, even if your `mkdir` kernel call won't create more than one level of directory at a time. `mkpath` takes three arguments:

- the name of the path to create, or a reference to a list of paths to create,
- a boolean value, which if `TRUE` will cause `mkpath` to print the name of each directory as it is created (defaults to `FALSE`), and
- the numeric mode to use when creating the directories (defaults to `0777`)

It returns a list of all directories (including intermediates, determined using the Unix `'/'` separator) created.

If a system error prevents a directory from being created, then the `mkpath` function throws a fatal error with `Carp::croak`. This error can be trapped with an `eval` block:

```
eval { mkpath($dir) };
if ($@) {
    print "Couldn't create $dir: $@";
}
```

Similarly, the `rmtree` function provides a convenient way to delete a subtree from the directory structure, much like the Unix command `rm -r`. `rmtree` takes three arguments:

- the root of the subtree to delete, or a reference to a list of roots. All of the files and directories below each root, as well as the roots themselves, will be deleted.
- a boolean value, which if `TRUE` will cause `rmtree` to print a message each time it examines a file, giving the name of the file, and indicating whether it's using `rmdir` or `unlink` to remove it, or that it's skipping it. (defaults to `FALSE`)
- a boolean value, which if `TRUE` will cause `rmtree` to skip any files to which you do not have delete access (if running under VMS) or write access (if running under another OS). This will change in the future when a criterion for 'delete permission' under OSs other than VMS is settled. (defaults to `FALSE`)

It returns the number of files successfully deleted. Symlinks are simply deleted and not followed.

NOTE: If the third parameter is not `TRUE`, `rmtree` is **unsecure** in the face of failure or interruption. Files and directories which were not deleted may be left with permissions reset to allow world read and write access. Note also that the occurrence of errors in `rmtree` can be determined *only* by trapping diagnostic messages using `$SIG{__WARN__}`; it is not apparent from the return value. Therefore, you must be extremely careful about using `rmtree($foo, $bar, 0` in situations where security is an issue.

AUTHORS

Tim Bunce <Tim.Bunce@ig.co.uk> and Charles Bailey <bailey@newman.upenn.edu>

NAME

File::Spec – portably perform operations on file names

SYNOPSIS

```
use File::Spec;

$x=File::Spec->catfile('a', 'b', 'c');
```

which returns 'a/b/c' under Unix. Or:

```
use File::Spec::Functions;

$x = catfile('a', 'b', 'c');
```

DESCRIPTION

This module is designed to support operations commonly performed on file specifications (usually called "file names", but not to be confused with the contents of a file, or Perl's file handles), such as concatenating several directory and file names into a single path, or determining whether a path is rooted. It is based on code directly taken from MakeMaker 5.17, code written by Andreas König, Andy Dougherty, Charles Bailey, Ilya Zakharevich, Paul Schinder, and others.

Since these functions are different for most operating systems, each set of OS specific routines is available in a separate module, including:

```
File::Spec::Unix
File::Spec::Mac
File::Spec::OS2
File::Spec::Win32
File::Spec::VMS
```

The module appropriate for the current OS is automatically loaded by File::Spec. Since some modules (like VMS) make use of facilities available only under that OS, it may not be possible to load all modules under all operating systems.

Since File::Spec is object oriented, subroutines should not be called directly, as in:

```
File::Spec::catfile('a', 'b');
```

but rather as class methods:

```
File::Spec->catfile('a', 'b');
```

For simple uses, *File::Spec::Functions* provides convenient functional forms of these methods.

METHODS

canonpath

No physical check on the filesystem, but a logical cleanup of a path.

```
$cpath = File::Spec->canonpath( $path );
```

catdir

Concatenate two or more directory names to form a complete path ending with a directory. But remove the trailing slash from the resulting string, because it doesn't look good, isn't necessary and confuses OS2. Of course, if this is the root directory, don't cut off the trailing slash :-)

```
$path = File::Spec->catdir( @directories );
```

catfile

Concatenate one or more directory names and a filename to form a complete path ending with a filename

```
$path = File::Spec->catfile( @directories, $filename );
```

curdir

Returns a string representation of the current directory.

```
$curdir = File::Spec->curdir();
```

devnull

Returns a string representation of the null device.

```
$devnull = File::Spec->devnull();
```

rootdir

Returns a string representation of the root directory.

```
$rootdir = File::Spec->rootdir();
```

tmpdir

Returns a string representation of the first writable directory from a list of possible temporary directories. Returns "" if no writable temporary directories are found. The list of directories checked depends on the platform; e.g. `File::Spec::Unix` checks `$ENV{TMPDIR}` and `/tmp`.

```
$tmpdir = File::Spec->tmpdir();
```

updir

Returns a string representation of the parent directory.

```
$updir = File::Spec->updir();
```

no_upwards

Given a list of file names, strip out those that refer to a parent directory. (Does not strip symlinks, only `'.'`, `'..'`, and equivalents.)

```
@paths = File::Spec->no_upwards( @paths );
```

case_tolerant

Returns a true or false value indicating, respectively, that alphabetic is not or is significant when comparing file specifications.

```
$is_case_tolerant = File::Spec->case_tolerant();
```

file_name_is_absolute

Takes as argument a path and returns true if it is an absolute path.

```
$is_absolute = File::Spec->file_name_is_absolute( $path );
```

This does not consult the local filesystem on Unix, Win32, OS/2, or Mac OS (Classic). It does consult the working environment for VMS (see [File::Spec::VMS/file_name_is_absolute](#)).

path

Takes no argument, returns the environment variable `PATH` as an array.

```
@PATH = File::Spec->path();
```

join

join is the same as `catfile`.

splitpath

Splits a path in to volume, directory, and filename portions. On systems with no concept of volume, returns undef for volume.

```
($volume,$directories,$file) = File::Spec->splitpath( $path );  
($volume,$directories,$file) = File::Spec->splitpath( $path, $no_file );
```

For systems with no syntax differentiating filenames from directories, assumes that the last file is a path unless `$no_file` is true or a trailing separator or `/.` or `./.` is present. On Unix this means that `$no_file` true makes this return (`"", $path, ""`).

The directory portion may or may not be returned with a trailing `'/'`.

The results can be passed to `/catpath()` to get back a path equivalent to (usually identical to) the original path.

splitdir

The opposite of `/catdir()`.

```
@dirs = File::Spec->splitdir( $directories );
```

`$directories` must be only the directory portion of the path on systems that have the concept of a volume or that have path syntax that differentiates files from directories.

Unlike just splitting the directories on the separator, empty directory names (`' '`) can be returned, because these are significant on some OSs.

catpath()

Takes volume, directory and file portions and returns an entire path. Under Unix, `$volume` is ignored, and directory and file are catenated. A `'/'` is inserted if need be. On other OSs, `$volume` is significant.

```
$full_path = File::Spec->catpath( $volume, $directory, $file );
```

abs2rel

Takes a destination path and an optional base path returns a relative path from the base path to the destination path:

```
$rel_path = File::Spec->abs2rel( $path ) ;
$rel_path = File::Spec->abs2rel( $path, $base ) ;
```

If `$base` is not present or `"`, then `cwd()` | Cwd is used. If `$base` is relative, then it is converted to absolute form using `/rel2abs()`. This means that it is taken to be relative to `cwd()` | Cwd.

On systems with the concept of a volume, this assumes that both paths are on the `$destination` volume, and ignores the `$base` volume.

On systems that have a grammar that indicates filenames, this ignores the `$base` filename as well. Otherwise all path components are assumed to be directories.

If `$path` is relative, it is converted to absolute form using `/rel2abs()`. This means that it is taken to be relative to `cwd()` | Cwd.

No checks against the filesystem are made. On VMS, there is interaction with the working environment, as logicals and macros are expanded.

Based on code written by Shigio Yamaguchi.

rel2abs()

Converts a relative path to an absolute path.

```
$abs_path = File::Spec->rel2abs( $path ) ;
$abs_path = File::Spec->rel2abs( $path, $base ) ;
```

If `$base` is not present or `"`, then `cwd()` | Cwd is used. If `$base` is relative, then it is converted to absolute form using `/rel2abs()`. This means that it is taken to be relative to `cwd()` | Cwd.

On systems with the concept of a volume, this assumes that both paths are on the `$base` volume, and ignores the `$path` volume.

On systems that have a grammar that indicates filenames, this ignores the `$base` filename as well. Otherwise all path components are assumed to be directories.

If `$path` is absolute, it is cleaned up and returned using `/canonpath()`.

No checks against the filesystem are made. On VMS, there is interaction with the working environment, as logicals and macros are expanded.

Based on code written by Shigio Yamaguchi.

For further information, please see *File::Spec::Unix*, *File::Spec::Mac*, *File::Spec::OS2*, *File::Spec::Win32*, or *File::Spec::VMS*.

SEE ALSO

File::Spec::Unix, *File::Spec::Mac*, *File::Spec::OS2*, *File::Spec::Win32*, *File::Spec::VMS*,
File::Spec::Functions, *ExtUtils::MakeMaker*

AUTHORS

Kenneth Albanowski <kjahds@kjahds.com>, Andy Dougherty <doughera@lafcol.lafayette.edu>, Andreas König <A.Koenig@franz.ww.TU-Berlin.DE>, Tim Bunce <Tim.Bunce@ig.co.uk>. VMS support by Charles Bailey <bailey@newman.upenn.edu>. OS/2 support by Ilya Zakharevich <ilya@math.ohio-state.edu>. Mac support by Paul Schinder <schinder@pobox.com>, and Thomas Wegner <wegner_thomas@yahoo.com>. `abs2rel()` and `rel2abs()` written by Shigio Yamaguchi <shigio@tamacom.com>, modified by Barrie Slaymaker <barries@slaysys.com>. `splitpath()`, `splitdir()`, `catpath()` and `catdir()` by Barrie Slaymaker.

NAME

File::Spec::Cygwin – methods for Cygwin file specs

SYNOPSIS

```
require File::Spec::Cygwin; # Done internally by File::Spec if needed
```

DESCRIPTION

See File::Spec::Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

This module is still in beta. Cygwin-knowledgeable folks are invited to offer patches and suggestions.

NAME

File::Spec::Epoc – methods for Epoc file specs

SYNOPSIS

```
require File::Spec::Epoc; # Done internally by File::Spec if needed
```

DESCRIPTION

See File::Spec::Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

This package is still work in progress ;-) o.flebbe@gmx.de

```
sub case_tolerant {  
    return 1;  
}
```

```
canonpath()
```

No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/.".

SEE ALSO

File::Spec

NAME

File::Spec::Functions – portably perform operations on file names

SYNOPSIS

```
use File::Spec::Functions;
$x = catfile('a', 'b');
```

DESCRIPTION

This module exports convenience functions for all of the class methods provided by File::Spec.

For a reference of available functions, please consult [File::Spec::Unix](#), which contains the entire set, and which is inherited by the modules for other platforms. For further information, please see [File::Spec::Mac](#), [File::Spec::OS2](#), [File::Spec::Win32](#), or [File::Spec::VMS](#).

Exports

The following functions are exported by default.

```
canonpath
catdir
catfile
curdir
rootdir
updir
no_upwards
file_name_is_absolute
path
```

The following functions are exported only by request.

```
devnull
tmpdir
splitpath
splitdir
catpath
abs2rel
rel2abs
```

All the functions may be imported using the `:ALL` tag.

SEE ALSO

File::Spec, File::Spec::Unix, File::Spec::Mac, File::Spec::OS2, File::Spec::Win32, File::Spec::VMS, ExtUtils::MakeMaker

NAME

File::Spec::Mac – File::Spec for Mac OS (Classic)

SYNOPSIS

```
require File::Spec::Mac; # Done internally by File::Spec if needed
```

DESCRIPTION

Methods for manipulating file specifications.

METHODS**canonpath**

On Mac OS, there's nothing to be done. Returns what it's given.

catdir()

Concatenate two or more directory names to form a path separated by colons (":") ending with a directory. Resulting paths are **relative** by default, but can be forced to be absolute (but avoid this, see below). Automatically puts a trailing ":" on the end of the complete path, because that's what's done in MacPerl's environment and helps to distinguish a file path from a directory path.

IMPORTANT NOTE: Beginning with version 1.3 of this module, the resulting path is relative by default and *not* absolute. This decision was made due to portability reasons. Since `File::Spec->catdir()` returns relative paths on all other operating systems, it will now also follow this convention on Mac OS. Note that this may break some existing scripts.

The intended purpose of this routine is to concatenate *directory names*. But because of the nature of Macintosh paths, some additional possibilities are allowed to make using this routine give reasonable results for some common situations. In other words, you are also allowed to concatenate *paths* instead of directory names (strictly speaking, a string like ":a" is a path, but not a name, since it contains a punctuation character ":").

So, beside calls like

```
catdir("a") = ":a:"
catdir("a","b") = ":a:b:"
catdir() = ""                (special case)
```

calls like the following

```
catdir(":a:") = ":a:"
catdir(":a","b") = ":a:b:"
catdir(":a:", "b") = ":a:b:"
catdir(":a:", ":b:") = ":a:b:"
catdir(":") = ":"
```

are allowed.

Here are the rules that are used in `catdir()`; note that we try to be as compatible as possible to Unix:

1. The resulting path is relative by default, i.e. the resulting path will have a leading colon.
2. A trailing colon is added automatically to the resulting path, to denote a directory.
3. Generally, each argument has one leading ":" and one trailing ":" removed (if any). They are then joined together by a ":". Special treatment applies for arguments denoting updir paths like "::lib:", see (4), or arguments consisting solely of colons ("colon paths"), see (5).

4.

When an updir path like ":::lib::" is passed as argument, the number of directories to climb up is handled correctly, not removing leading or trailing colons when necessary. E.g.

```
catdir(":::a", ":::b", "c")    = ":::a::b:c:"
catdir(":::a::", ":::b", "c")  = ":::a:::b:c:"
```

5.

Adding a colon ":" or empty string "" to a path at *any* position doesn't alter the path, i.e. these arguments are ignored. (When a "" is passed as the first argument, it has a special meaning, see (6)). This way, a colon ":" is handled like a "." (curdir) on Unix, while an empty string "" is generally ignored (see `Unix->canonpath()`). Likewise, a "::" is handled like a ".." (updir), and a ":::" is handled like a ".../" etc. E.g.

```
catdir("a", ":", ":", "b")     = ":a:b:"
catdir("a", ":", ":", "b")    = ":a::b:"
```

6.

If the first argument is an empty string "" or is a volume name, i.e. matches the pattern `/^[^:]+/`, the resulting path is **absolute**.

7.

Passing an empty string "" as the first argument to `catdir()` is like passing `File::Spec->rootdir()` as the first argument, i.e.

```
catdir("", "a", "b")           is the same as
catdir(rootdir(), "a", "b").
```

This is true on Unix, where `catdir("", "a", "b")` yields `/a/b` and `rootdir()` is `/`. Note that `rootdir()` on Mac OS is the startup volume, which is the closest in concept to Unix' `/`. This should help to run existing scripts originally written for Unix.

8.

For absolute paths, some cleanup is done, to ensure that the volume name isn't immediately followed by updirs. This is invalid, because this would go beyond "root". Generally, these cases are handled like their Unix counterparts:

Unix:

```
Unix->catdir("", "")           = "/"
Unix->catdir("", ".")          = "/"
Unix->catdir("", "..")         = "/" # can't go beyond root
Unix->catdir("", ".", "..", "..", "a") = "/a"
```

Mac:

```
Mac->catdir("", "")           = rootdir()      # (e.g. "HD:")
Mac->catdir("", ":")          = rootdir()
Mac->catdir("", "::")          = rootdir()      # can't go beyond root
Mac->catdir("", ":", ":", "a") = rootdir() . "a:" # (e.g. "HD:a:")
```

However, this approach is limited to the first arguments following "root" (again, see `Unix->canonpath()`). If there are more arguments that move up the directory tree, an invalid path going beyond root can be created.

As you've seen, you can force `catdir()` to create an absolute path by passing either an empty string or a path that begins with a volume name as the first argument. However, you are strongly encouraged not to do so, since this is done only for backward compatibility. Newer versions of `File::Spec` come with a method called `catpath()` (see below), that is designed to offer a portable solution for the creation of absolute paths. It takes volume, directory and file portions and returns an entire path. While `catdir()` is still suitable for the concatenation of *directory names*, you are encouraged to use `catpath()` to

concatenate *volume names* and *directory paths*. E.g.

```
$dir      = File::Spec->catdir("tmp", "sources");
$abs_path = File::Spec->catpath("MacintoshHD:", $dir, "");
```

yields

```
"MacintoshHD:tmp:sources:" .
```

catfile

Concatenate one or more directory names and a filename to form a complete path ending with a filename. Resulting paths are **relative** by default, but can be forced to be absolute (but avoid this).

IMPORTANT NOTE: Beginning with version 1.3 of this module, the resulting path is relative by default and *not* absolute. This decision was made due to portability reasons. Since `File::Spec->catfile()` returns relative paths on all other operating systems, it will now also follow this convention on Mac OS. Note that this may break some existing scripts.

The last argument is always considered to be the file portion. Since `catfile()` uses `catdir()` (see above) for the concatenation of the directory portions (if any), the following with regard to relative and absolute paths is true:

```
catfile("")      = ""
catfile("file") = "file"
```

but

```
catfile("", "")      = rootdir()      # (e.g. "HD:")
catfile("", "file")  = rootdir() . file # (e.g. "HD:file")
catfile("HD:", "file") = "HD:file"
```

This means that `catdir()` is called only when there are two or more arguments, as one might expect.

Note that the leading ":" is removed from the filename, so that

```
catfile("a", "b", "file") = ":a:b:file"    and
catfile("a", "b", ":file") = ":a:b:file"
```

give the same answer.

To concatenate *volume names*, *directory paths* and *filenames*, you are encouraged to use `catpath()` (see below).

curdir

Returns a string representing the current directory. On Mac OS, this is ":".

devnull

Returns a string representing the null device. On Mac OS, this is "Dev:Null".

rootdir

Returns a string representing the root directory. Under MacPerl, returns the name of the startup volume, since that's the closest in concept, although other volumes aren't rooted there. The name has a trailing ":", because that's the correct specification for a volume name on Mac OS.

If `Mac::Files` could not be loaded, the empty string is returned.

tmpdir

Returns the contents of `$ENV{TMPDIR}`, if that directory exists or the current working directory otherwise. Under MacPerl, `$ENV{TMPDIR}` will contain a path like "MacintoshHD:Temporary Items:", which is a hidden directory on your startup volume.

updir

Returns a string representing the parent directory. On Mac OS, this is ":".

file_name_is_absolute

Takes as argument a path and returns true, if it is an absolute path. If the path has a leading ":", it's a relative path. Otherwise, it's an absolute path, unless the path doesn't contain any colons, i.e. it's a name like "a". In this particular case, the path is considered to be relative (i.e. it is considered to be a filename). Use ":" in the appropriate place in the path if you want to distinguish unambiguously. As a special case, the filename "" is always considered to be absolute. Note that with version 1.2 of File::Spec::Mac, this does no longer consult the local filesystem.

E.g.

```
File::Spec->file_name_is_absolute("a");           # false (relative)
File::Spec->file_name_is_absolute(":a:b:");       # false (relative)
File::Spec->file_name_is_absolute("MacintoshHD:"); # true (absolute)
File::Spec->file_name_is_absolute("");           # true (absolute)
```

path

Returns the null list for the MacPerl application, since the concept is usually meaningless under Mac OS. But if you're using the MacPerl tool under MPW, it gives back \$ENV{Commands} suitably split, as is done in :lib:ExtUtils:MM_Mac.pm.

splitpath

```
($volume,$directories,$file) = File::Spec->splitpath( $path );
($volume,$directories,$file) = File::Spec->splitpath( $path, $no_file );
```

Splits a path in to volume, directory, and filename portions.

On Mac OS, assumes that the last part of the path is a filename unless \$no_file is true or a trailing separator ":" is present.

The volume portion is always returned with a trailing ":". The directory portion is always returned with a leading (to denote a relative path) and a trailing ":" (to denote a directory). The file portion is always returned *without* a leading ":". Empty portions are returned as empty string "".

The results can be passed to catpath() to get back a path equivalent to (usually identical to) the original path.

splitdir

The opposite of catdir().

```
@dirs = File::Spec->splitdir( $directories );
```

\$directories should be only the directory portion of the path on systems that have the concept of a volume or that have path syntax that differentiates files from directories. Consider using splitpath() otherwise.

Unlike just splitting the directories on the separator, empty directory names ("") can be returned. Since catdir() on Mac OS always appends a trailing colon to distinguish a directory path from a file path, a single trailing colon will be ignored, i.e. there's no empty directory name after it.

Hence, on Mac OS, both

```
File::Spec->splitdir( ":a:b::c:" );    and
File::Spec->splitdir( ":a:b::c" );
```

yield:

```
( "a", "b", ":", "c")
```

while

```
File::Spec->splitdir( ":a:b::c:" );
```

yields:

```
( "a", "b", ":", "c", ":")
```

catpath

```
$path = File::Spec->catpath($volume,$directory,$file);
```

Takes volume, directory and file portions and returns an entire path. On Mac OS, \$volume, \$directory and \$file are concatenated. A ‘.’ is inserted if need be. You may pass an empty string for each portion. If all portions are empty, the empty string is returned. If \$volume is empty, the result will be a relative path, beginning with a ‘.’. If \$volume and \$directory are empty, a leading ":" (if any) is removed from \$file and the remainder is returned. If \$file is empty, the resulting path will have a trailing ‘.’.

abs2rel

Takes a destination path and an optional base path and returns a relative path from the base path to the destination path:

```
$rel_path = File::Spec->abs2rel( $path ) ;
$rel_path = File::Spec->abs2rel( $path, $base ) ;
```

Note that both paths are assumed to have a notation that distinguishes a directory path (with trailing ‘.’) from a file path (without trailing ‘.’).

If \$base is not present or ‘’, then the current working directory is used. If \$base is relative, then it is converted to absolute form using `rel2abs()`. This means that it is taken to be relative to the current working directory.

Since Mac OS has the concept of volumes, this assumes that both paths are on the \$destination volume, and ignores the \$base volume (!).

If \$base doesn’t have a trailing colon, the last element of \$base is assumed to be a filename. This filename is ignored (!). Otherwise all path components are assumed to be directories.

If \$path is relative, it is converted to absolute form using `rel2abs()`. This means that it is taken to be relative to the current working directory.

Based on code written by Shigio Yamaguchi.

rel2abs

Converts a relative path to an absolute path:

```
$abs_path = File::Spec->rel2abs( $path ) ;
$abs_path = File::Spec->rel2abs( $path, $base ) ;
```

Note that both paths are assumed to have a notation that distinguishes a directory path (with trailing ‘.’) from a file path (without trailing ‘.’).

If \$base is not present or ‘’, then \$base is set to the current working directory. If \$base is relative, then it is converted to absolute form using `rel2abs()`. This means that it is taken to be relative to the current working directory.

If \$base doesn’t have a trailing colon, the last element of \$base is assumed to be a filename. This filename is ignored (!). Otherwise all path components are assumed to be directories.

If \$path is already absolute, it is returned and \$base is ignored.

Based on code written by Shigio Yamaguchi.

AUTHORS

See the authors list in *File::Spec*. Mac OS support by Paul Schinder <schinder@pobox.com> and Thomas Wegner <wegner_thomas@yahoo.com>.

SEE ALSO

File::Spec

canonpath

No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/".

splitpath

```
($volume,$directories,$file) = File::Spec->splitpath( $path );
($volume,$directories,$file) = File::Spec->splitpath( $path, $no_file );
```

Splits a path in to volume, directory, and filename portions. Assumes that the last file is a path unless the path ends in '/', './', or '..' or `$no_file` is true. On Win32 this means that `$no_file` true makes this return (`$volume, $path, undef`).

Separators accepted are \ and /.

Volumes can be drive letters or UNC sharenames (\\server\share).

The results can be passed to [/catpath](#) to get back a path equivalent to (usually identical to) the original path.

splitdir

The opposite of [catdir\(\)](#).

```
@dirs = File::Spec->splitdir( $directories );
```

`$directories` must be only the directory portion of the path on systems that have the concept of a volume or that have path syntax that differentiates files from directories.

Unlike just splitting the directories on the separator, leading empty and trailing directory entries can be returned, because these are significant on some OSs. So,

```
File::Spec->splitdir( "/a/b//c/" );
```

Yields:

```
( '', 'a', 'b', '', 'c', '' )
```

catpath

Takes volume, directory and file portions and returns an entire path. Under Unix, `$volume` is ignored, and this is just like `catfile()`. On other OSs, the `$volume` become significant.

NAME

File::Spec::OS2 – methods for OS/2 file specs

SYNOPSIS

```
require File::Spec::OS2; # Done internally by File::Spec if needed
```

DESCRIPTION

See `File::Spec::Unix` for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

NAME

File::Spec::Unix – File::Spec for Unix, base for other File::Spec modules

SYNOPSIS

```
require File::Spec::Unix; # Done automatically by File::Spec
```

DESCRIPTION

Methods for manipulating file specifications. Other File::Spec modules, such as File::Spec::Mac, inherit from File::Spec::Unix and override specific methods.

METHODS

`canonpath()`

No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminates successive slashes and successive `"/."`.

```
$cpath = File::Spec->canonpath( $path ) ;
```

`catdir()`

Concatenate two or more directory names to form a complete path ending with a directory. But remove the trailing slash from the resulting string, because it doesn't look good, isn't necessary and confuses OS2. Of course, if this is the root directory, don't cut off the trailing slash :-)

`catfile`

Concatenate one or more directory names and a filename to form a complete path ending with a filename

`curdir`

Returns a string representation of the current directory. `"/."` on UNIX.

`devnull`

Returns a string representation of the null device. `"/dev/null"` on UNIX.

`rootdir`

Returns a string representation of the root directory. `"/"` on UNIX.

`tmpdir`

Returns a string representation of the first writable directory from the following list or `""` if none are writable:

```
$ENV{TMPDIR}  
/tmp
```

Since perl 5.8.0, if running under taint mode, and if `$ENV{TMPDIR}` is tainted, it is not used.

`updir`

Returns a string representation of the parent directory. `"/.."` on UNIX.

`no_upwards`

Given a list of file names, strip out those that refer to a parent directory. (Does not strip symlinks, only `'.'`, `'..'`, and equivalents.)

`case_tolerant`

Returns a true or false value indicating, respectively, that alphabetic is not or is significant when comparing file specifications.

`file_name_is_absolute`

Takes as argument a path and returns true if it is an absolute path.

This does not consult the local filesystem on Unix, Win32, OS/2 or Mac OS (Classic). It does consult the working environment for VMS (see [File::Spec::VMS/file_name_is_absolute](#)).

path

Takes no argument, returns the environment variable `PATH` as an array.

join

`join` is the same as `catfile`.

splitpath

```
($volume,$directories,$file) = File::Spec->splitpath( $path );
($volume,$directories,$file) = File::Spec->splitpath( $path, $no_file );
```

Splits a path in to volume, directory, and filename portions. On systems with no concept of volume, returns `undef` for volume.

For systems with no syntax differentiating filenames from directories, assumes that the last file is a path unless `$no_file` is true or a trailing separator or `/.` or `./.` is present. On Unix this means that `$no_file` true makes this return `("", $path, "")`.

The directory portion may or may not be returned with a trailing `'/'`.

The results can be passed to `/catpath()` to get back a path equivalent to (usually identical to) the original path.

splitdir

The opposite of `/catdir()`.

```
@dirs = File::Spec->splitdir( $directories );
```

`$directories` must be only the directory portion of the path on systems that have the concept of a volume or that have path syntax that differentiates files from directories.

Unlike just splitting the directories on the separator, empty directory names `('')` can be returned, because these are significant on some OSs.

On Unix,

```
File::Spec->splitdir( "/a/b//c/" );
```

Yields:

```
( '', 'a', 'b', '', 'c', '' )
```

catpath()

Takes volume, directory and file portions and returns an entire path. Under Unix, `$volume` is ignored, and directory and file are catenated. A `'/'` is inserted if needed (though if the directory portion doesn't start with `'/'` it is not added). On other OSs, `$volume` is significant.

abs2rel

Takes a destination path and an optional base path returns a relative path from the base path to the destination path:

```
$rel_path = File::Spec->abs2rel( $path );
$rel_path = File::Spec->abs2rel( $path, $base );
```

If `$base` is not present or `""`, then `cwd()` | `Cwd` is used. If `$base` is relative, then it is converted to absolute form using `/rel2abs()`. This means that it is taken to be relative to `cwd()` | `Cwd`.

On systems with the concept of a volume, this assumes that both paths are on the `$destination` volume, and ignores the `$base` volume.

On systems that have a grammar that indicates filenames, this ignores the `$base` filename as well. Otherwise all path components are assumed to be directories.

If `$path` is relative, it is converted to absolute form using `/rel2abs()`. This means that it is taken to

be relative to [cwd\(\)](#) | Cwd.

No checks against the filesystem are made. On VMS, there is interaction with the working environment, as logicals and macros are expanded.

Based on code written by Shigio Yamaguchi.

`rel2abs()`

Converts a relative path to an absolute path.

```
$abs_path = File::Spec->rel2abs( $path ) ;  
$abs_path = File::Spec->rel2abs( $path, $base ) ;
```

If `$base` is not present or `''`, then [cwd\(\)](#) | Cwd is used. If `$base` is relative, then it is converted to absolute form using [/rel2abs\(\)](#). This means that it is taken to be relative to [cwd\(\)](#) | Cwd.

On systems with the concept of a volume, this assumes that both paths are on the `$base` volume, and ignores the `$path` volume.

On systems that have a grammar that indicates filenames, this ignores the `$base` filename as well. Otherwise all path components are assumed to be directories.

If `$path` is absolute, it is cleaned up and returned using [/canonpath\(\)](#).

No checks against the filesystem are made. On VMS, there is interaction with the working environment, as logicals and macros are expanded.

Based on code written by Shigio Yamaguchi.

SEE ALSO

[File::Spec](#)

NAME

File::Spec::VMS – methods for VMS file specs

SYNOPSIS

```
require File::Spec::VMS; # Done internally by File::Spec if needed
```

DESCRIPTION

See File::Spec::Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

eliminate_macros

Expands MM[KS]/Make macros in a text string, using the contents of identically named elements of `$_$self`, and returns the result as a file specification in Unix syntax.

fixpath

Catchall routine to clean up problem MM[SK]/Make macros. Expands macros in any directory specification, in order to avoid juxtaposing two VMS-syntax directories when MM[SK] is run. Also expands expressions which are all macro, so that we can tell how long the expansion is, and avoid overrunning DCL's command buffer when MM[KS] is running.

If optional second argument has a TRUE value, then the return string is a VMS-syntax directory specification, if it is FALSE, the return string is a VMS-syntax file specification, and if it is not specified, `fixpath()` checks to see whether it matches the name of a directory in the current default directory, and returns a directory or file specification accordingly.

Methods always loaded**canonpath (override)**

Removes redundant portions of file specifications according to VMS syntax.

catdir

Concatenates a list of file specifications, and returns the result as a VMS-syntax directory specification. No check is made for "impossible" cases (e.g. elements other than the first being absolute filespecs).

catfile

Concatenates a list of file specifications, and returns the result as a VMS-syntax file specification.

curdir (override)

Returns a string representation of the current directory: `[']`

devnull (override)

Returns a string representation of the null device: `['_NLA0:']`

rootdir (override)

Returns a string representation of the root directory: `['SYS$DISK: [000000]']`

tmpdir (override)

Returns a string representation of the first writable directory from the following list or `''` if none are writable:

```
sys$scratch:
$ENV{TMPDIR}
```

Since perl 5.8.0, if running under taint mode, and if `$ENV{TMPDIR}` is tainted, it is not used.

updir (override)

Returns a string representation of the parent directory: `['-']`

`case_tolerant` (override)

VMS file specification syntax is case-tolerant.

`path` (override)

Translate logical name `DCL$PATH` as a searchlist, rather than trying to `split` string value of `$ENV{ 'PATH' }`.

`file_name_is_absolute` (override)

Checks for VMS directory spec as well as Unix separators.

`splitpath` (override)

Splits using VMS syntax.

`splitdir` (override)

Split dirs spec using VMS syntax.

`catpath` (override)

Construct a complete filespec using VMS syntax

`abs2rel` (override)

Use VMS syntax when converting filespecs.

`rel2abs` (override)

Use VMS syntax when converting filespecs.

SEE ALSO

[File::Spec](#)

NAME

File::Spec::Win32 – methods for Win32 file specs

SYNOPSIS

```
require File::Spec::Win32; # Done internally by File::Spec if needed
```

DESCRIPTION

See File::Spec::Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

devnull

Returns a string representation of the null device.

tmpdir

Returns a string representation of the first existing directory from the following list:

```
$ENV{TMPDIR}
$ENV{TEMP}
$ENV{TMP}
SYS:/temp
C:/temp
/tmp
/
```

The SYS:/temp is preferred in Novell NetWare.

Since Perl 5.8.0, if running under taint mode, and if the environment variables are tainted, they are not used.

catfile

Concatenate one or more directory names and a filename to form a complete path ending with a filename

canonpath

No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/.".

splitpath

```
($volume,$directories,$file) = File::Spec->splitpath( $path );
($volume,$directories,$file) = File::Spec->splitpath( $path, $no_file );
```

Splits a path in to volume, directory, and filename portions. Assumes that the last file is a path unless the path ends in '\\', '\\.', '\\..' or \$no_file is true. On Win32 this means that \$no_file true makes this return (\$volume, \$path, undef).

Separators accepted are \ and /.

Volumes can be drive letters or UNC sharenames (\\server\share).

The results can be passed to [/catpath](#) to get back a path equivalent to (usually identical to) the original path.

splitdir

The opposite of [catdir\(\)](#).

```
@dirs = File::Spec->splitdir( $directories );
```

\$directories must be only the directory portion of the path on systems that have the concept of a volume or that have path syntax that differentiates files from directories.

Unlike just splitting the directories on the separator, leading empty and trailing directory entries can

be returned, because these are significant on some OSs. So,

```
File::Spec->splitdir( "/a/b/c" );
```

Yields:

```
( ' ', 'a', 'b', ' ', 'c', ' ' )
```

`catpath`

Takes volume, directory and file portions and returns an entire path. Under Unix, `$volume` is ignored, and this is just like `catfile()`. On other OSs, the `$volume` become significant.

Note For File::Spec::Win32 Maintainers

Novell NetWare inherits its `File::Spec` behaviour from `File::Spec::Win32`.

SEE ALSO

[File::Spec](#)

NAME

File::stat – by-name interface to Perl's built-in `stat()` functions

SYNOPSIS

```
use File::stat;
$st = stat($file) or die "No $file: $!";
if ( ($st->mode & 0111) && $st->nlink > 1 ) {
    print "$file is executable with lotsa links\n";
}

use File::stat qw(:FIELDS);
stat($file) or die "No $file: $!";
if ( ($st_mode & 0111) && $st_nlink > 1 ) {
    print "$file is executable with lotsa links\n";
}
```

DESCRIPTION

This module's default exports override the core `stat()` and `lstat()` functions, replacing them with versions that return "File::stat" objects. This object has methods that return the similarly named structure field name from the `stat(2)` function; namely, `dev`, `ino`, `mode`, `nlink`, `uid`, `gid`, `rdev`, `size`, `atime`, `mtime`, `ctime`, `blksize`, and `blocks`.

You may also import all the structure fields directly into your namespace as regular variables using the `:FIELDS` import tag. (Note that this still overrides your `stat()` and `lstat()` functions.) Access these fields as variables named with a preceding `st_` in front their method names. Thus, `$stat_obj->dev()` corresponds to `$st_dev` if you import the fields.

To access this functionality without the core overrides, pass the use an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::pseudo-package`.

BUGS

As of Perl 5.8.0 after using this module you cannot use the implicit `$_` or the special filehandle `_` with `stat()` or `lstat()`, trying to do so leads into strange errors. The workaround is for `$_` to be explicit

```
my $stat_obj = stat $_;
```

and for `_` to explicitly populate the object using the unexported and undocumented `populate()` function with `CORE::stat()`:

```
my $stat_obj = File::stat::populate(CORE::stat(_));
```

NOTE

While this class is currently implemented using the `Class::Struct` module to build a struct-like class, you shouldn't rely upon this.

AUTHOR

Tom Christiansen

NAME

File::Temp – return name and handle of a temporary file safely

=begin __INTERNALS

PORTABILITY

This module is designed to be portable across operating systems and it currently supports Unix, VMS, DOS, OS/2, Windows and Mac OS (Classic). When porting to a new OS there are generally three main issues that have to be solved:

- Can the OS unlink an open file? If it can not then the `_can_unlink_opened_file` method should be modified.
- Are the return values from `stat` reliable? By default all the return values from `stat` are compared when unlinking a temporary file using the filename and the handle. Operating systems other than unix do not always have valid entries in all fields. If `unlink0` fails then the `stat` comparison should be modified accordingly.
- Security. Systems that can not support a test for the sticky bit on a directory can not use the MEDIUM and HIGH security tests. The `_can_do_level` method should be modified accordingly.

=end __INTERNALS

SYNOPSIS

```
use File::Temp qw/ tempfile tempdir /;

$dir = tempdir( CLEANUP => 1 );
($fh, $filename) = tempfile( DIR => $dir );

($fh, $filename) = tempfile( $template, DIR => $dir );
($fh, $filename) = tempfile( $template, SUFFIX => '.dat' );

$fh = tempfile();
```

MkTemp family:

```
use File::Temp qw/ :mktemp /;

($fh, $file) = mkstemp( "tmpfileXXXXXX" );
($fh, $file) = mkstemp( "tmpfileXXXXXX", $suffix );

$tmpdir = mkdtemp( $template );

$unopened_file = mktemp( $template );
```

POSIX functions:

```
use File::Temp qw/ :POSIX /;

$file = tmpnam();
$fh = tmpfile();

($fh, $file) = tmpnam();
($fh, $file) = tmpfile();
```

Compatibility functions:

```
$unopened_file = File::Temp::tempnam( $dir, $pfx );
```

=begin later

Objects (NOT YET IMPLEMENTED):

```
require File::Temp;
```

```
$fh = new File::Temp($template);
$fname = $fh->filename;

=end later
```

DESCRIPTION

`File::Temp` can be used to create and open temporary files in a safe way. The `tempfile()` function can be used to return the name and the open filehandle of a temporary file. The `tmpdir()` function can be used to create a temporary directory.

The security aspect of temporary file creation is emphasized such that a filehandle and filename are returned together. This helps guarantee that a race condition can not occur where the temporary file is created by another process between checking for the existence of the file and its opening. Additional security levels are provided to check, for example, that the sticky bit is set on world writable directories. See *"safe_level"* for more information.

For compatibility with popular C library functions, Perl implementations of the `mkstemp()` family of functions are provided. These are, `mkstemp()`, `mkstemps()`, `mkdtemp()` and `mktemp()`.

Additionally, implementations of the standard *POSIX/POSIX* `tmpnam()` and `tmpfile()` functions are provided if required.

Implementations of `mktemp()`, `tmpnam()`, and `tempnam()` are provided, but should be used with caution since they return only a filename that was valid when function was called, so cannot guarantee that the file will not exist by the time the caller opens the filename.

FUNCTIONS

This section describes the recommended interface for generating temporary files and directories.

tempfile

This is the basic function to generate temporary files. The behaviour of the file can be changed using various options:

```
($fh, $filename) = tempfile();
```

Create a temporary file in the directory specified for temporary files, as specified by the `tmpdir()` function in *File::Spec*.

```
($fh, $filename) = tempfile($template);
```

Create a temporary file in the current directory using the supplied template. Trailing 'X' characters are replaced with random letters to generate the filename. At least four 'X' characters must be present in the template.

```
($fh, $filename) = tempfile($template, SUFFIX => $suffix)
```

Same as previously, except that a suffix is added to the template after the 'X' translation. Useful for ensuring that a temporary filename has a particular extension when needed by other applications. But see the WARNING at the end.

```
($fh, $filename) = tempfile($template, DIR => $dir);
```

Translates the template as before except that a directory name is specified.

```
($fh, $filename) = tempfile($template, UNLINK => 1);
```

Return the filename and filehandle as before except that the file is automatically removed when the program exits. Default is for the file to be removed if a file handle is requested and to be kept if the filename is requested. In a scalar context (where no filename is returned) the file is always deleted either on exit or when it is closed.

If the template is not specified, a template is always automatically generated. This temporary file is placed in `tmpdir()` (*File::Spec*) unless a directory is specified explicitly with the `DIR` option.

```
$fh = tempfile( $template, DIR => $dir );
```

If called in scalar context, only the filehandle is returned and the file will automatically be deleted when closed (see the description of `tempfile()` elsewhere in this document). This is the preferred mode of operation, as if you only have a filehandle, you can never create a race condition by fumbling with the filename. On systems that can not unlink an open file or can not mark a file as temporary when it is opened (for example, Windows NT uses the `O_TEMPORARY` flag)) the file is marked for deletion when the program ends (equivalent to setting `UNLINK` to 1). The `UNLINK` flag is ignored if present.

```
(undef, $filename) = tempfile($template, OPEN => 0);
```

This will return the filename based on the template but will not open this file. Cannot be used in conjunction with `UNLINK` set to true. Default is to always open the file to protect from possible race conditions. A warning is issued if warnings are turned on. Consider using the `tmpnam()` and `mktemp()` functions described elsewhere in this document if opening the file is not required.

Options can be combined as required.

tmpdir

This is the recommended interface for creation of temporary directories. The behaviour of the function depends on the arguments:

```
$tmpdir = tmpdir();
```

Create a directory in `tmpdir()` (see [File::Spec/File::Spec](#)).

```
$tmpdir = tmpdir( $template );
```

Create a directory from the supplied template. This template is similar to that described for `tempfile()`. 'X' characters at the end of the template are replaced with random letters to construct the directory name. At least four 'X' characters must be in the template.

```
$tmpdir = tmpdir ( DIR => $dir );
```

Specifies the directory to use for the temporary directory. The temporary directory name is derived from an internal template.

```
$tmpdir = tmpdir ( $template, DIR => $dir );
```

Prepend the supplied directory name to the template. The template should not include parent directory specifications itself. Any parent directory specifications are removed from the template before prepending the supplied directory.

```
$tmpdir = tmpdir ( $template, TMPDIR => 1 );
```

Using the supplied template, create the temporary directory in a standard location for temporary files. Equivalent to doing

```
$tmpdir = tmpdir ( $template, DIR => File::Spec->tmpdir);
```

but shorter. Parent directory specifications are stripped from the template itself. The `TMPDIR` option is ignored if `DIR` is set explicitly. Additionally, `TMPDIR` is implied if neither a template nor a directory are supplied.

```
$tmpdir = tmpdir( $template, CLEANUP => 1);
```

Create a temporary directory using the supplied template, but attempt to remove it (and all files inside it) when the program exits. Note that an attempt will be made to remove all files from the directory even if they were not created by this module (otherwise why ask to clean it up?). The directory removal is made with the `rmtree()` function from the [File::Path/File::Path](#) module. Of course, if the template is not specified, the temporary directory will be created in `tmpdir()` and will also be removed at program exit.

MKTEMP FUNCTIONS

The following functions are Perl implementations of the `mktemp()` family of temp file generation system calls.

mkstemp

Given a template, returns a filehandle to the temporary file and the name of the file.

```
($fh, $name) = mkstemp( $template );
```

In scalar context, just the filehandle is returned.

The template may be any filename with some number of X's appended to it, for example */tmp/temp.XXXX*. The trailing X's are replaced with unique alphanumeric combinations.

mkstemps

Similar to `mkstemp()`, except that an extra argument can be supplied with a suffix to be appended to the template.

```
($fh, $name) = mkstemps( $template, $suffix );
```

For example a template of `testXXXXXX` and suffix of `.dat` would generate a file similar to *testhGji_w.dat*.

Returns just the filehandle alone when called in scalar context.

mkdtemp

Create a directory from a template. The template must end in X's that are replaced by the routine.

```
$tmpdir_name = mkdtemp($template);
```

Returns the name of the temporary directory created. Returns undef on failure.

Directory must be removed by the caller.

mktemp

Returns a valid temporary filename but does not guarantee that the file will not be opened by someone else.

```
$unopened_file = mktemp($template);
```

Template is the same as that required by `mkstemp()`.

POSIX FUNCTIONS

This section describes the re-implementation of the `tmpnam()` and `tmpfile()` functions described in [POSIX](#) using the `mkstemp()` from this module.

Unlike the [POSIX/POSIX](#) implementations, the directory used for the temporary file is not specified in a system include file (`P_tmpdir`) but simply depends on the choice of `tmpdir()` returned by [File::Spec/File::Spec](#). On some implementations this location can be set using the `TMPDIR` environment variable, which may not be secure. If this is a problem, simply use `mkstemp()` and specify a template.

tmpnam

When called in scalar context, returns the full name (including path) of a temporary file (uses `mktemp()`). The only check is that the file does not already exist, but there is no guarantee that that condition will continue to apply.

```
$file = tmpnam();
```

When called in list context, a filehandle to the open file and a filename are returned. This is achieved by calling `mkstemp()` after constructing a suitable template.

```
($fh, $file) = tmpnam();
```

If possible, this form should be used to prevent possible race conditions.

See [File::Spec/tmpdir](#) for information on the choice of temporary directory for a particular operating system.

tmpfile

In scalar context, returns the filehandle of a temporary file.

```
$fh = tmpfile();
```

The file is removed when the filehandle is closed or when the program exits. No access to the filename is provided.

If the temporary file can not be created undef is returned. Currently this command will probably not work when the temporary directory is on an NFS file system.

ADDITIONAL FUNCTIONS

These functions are provided for backwards compatibility with common tempfile generation C library functions.

They are not exported and must be addressed using the full package name.

tempnam

Return the name of a temporary file in the specified directory using a prefix. The file is guaranteed not to exist at the time the function was called, but such guarantees are good for one clock tick only. Always use the proper form of `sysopen` with `O_CREAT | O_EXCL` if you must open such a filename.

```
$filename = File::Temp::tempnam( $dir, $prefix );
```

Equivalent to running `mktemp()` with `$dir/$prefixXXXXXXXX` (using unix file convention as an example)

Because this function uses `mktemp()`, it can suffer from race conditions.

UTILITY FUNCTIONS

Useful functions for dealing with the filehandle and filename.

unlink0

Given an open filehandle and the associated filename, make a safe unlink. This is achieved by first checking that the filename and filehandle initially point to the same file and that the number of links to the file is 1 (all fields returned by `stat()` are compared). Then the filename is unlinked and the filehandle checked once again to verify that the number of links on that file is now 0. This is the closest you can come to making sure that the filename unlinked was the same as the file whose descriptor you hold.

```
unlink0($fh, $path) or die "Error unlinking file $path safely";
```

Returns false on error. The filehandle is not closed since on some occasions this is not required.

On some platforms, for example Windows NT, it is not possible to unlink an open file (the file must be closed first). On those platforms, the actual unlinking is deferred until the program ends and good status is returned. A check is still performed to make sure that the filehandle and filename are pointing to the same thing (but not at the time the end block is executed since the deferred removal may not have access to the filehandle).

Additionally, on Windows NT not all the fields returned by `stat()` can be compared. For example, the `dev` and `rdev` fields seem to be different. Also, it seems that the size of the file returned by `stat()` does not always agree, with `stat(FH)` being more accurate than `stat(filename)`, presumably because of caching issues even when using `autoflush` (this is usually overcome by waiting a while after writing to the tempfile before attempting to `unlink0` it).

Finally, on NFS file systems the link count of the file handle does not always go to zero immediately after unlinking. Currently, this command is expected to fail on NFS disks.

PACKAGE VARIABLES

These functions control the global state of the package.

safe_level

Controls the lengths to which the module will go to check the safety of the temporary file or directory before proceeding. Options are:

STANDARD

Do the basic security measures to ensure the directory exists and is writable, that the `umask()` is fixed before opening of the file, that temporary files are opened only if they do not already exist, and that possible race conditions are avoided. Finally the `unlink0/unlink0` function is used to remove files safely.

MEDIUM In addition to the STANDARD security, the output directory is checked to make sure that it is owned either by root or the user running the program. If the directory is writable by group or by other, it is then checked to make sure that the sticky bit is set.

Will not work on platforms that do not support the `-k` test for sticky bit.

HIGH In addition to the MEDIUM security checks, also check for the possibility of ```chown()` giveaway'' using the *POSIX/POSIX* `sysconf()` function. If this is a possibility, each directory in the path is checked in turn for safeness, recursively walking back to the root directory.

For platforms that do not support the *POSIX/POSIX* `_PC_CHOWN_RESTRICTED` symbol (for example, Windows NT) it is assumed that ```chown()` giveaway'' is possible and the recursive test is performed.

The level can be changed as follows:

```
File::Temp->safe_level( File::Temp::HIGH );
```

The level constants are not exported by the module.

Currently, you must be running at least perl v5.6.0 in order to run with MEDIUM or HIGH security. This is simply because the safety tests use functions from *Fcntl/Fcntl* that are not available in older versions of perl. The problem is that the version number for Fcntl is the same in perl 5.6.0 and in 5.005_03 even though they are different versions.

On systems that do not support the HIGH or MEDIUM safety levels (for example Win NT or OS/2) any attempt to change the level will be ignored. The decision to ignore rather than raise an exception allows portable programs to be written with high security in mind for the systems that can support this without those programs failing on systems where the extra tests are irrelevant.

If you really need to see whether the change has been accepted simply examine the return value of `safe_level`.

```
$newlevel = File::Temp->safe_level( File::Temp::HIGH );
die "Could not change to high security"
    if $newlevel != File::Temp::HIGH;
```

TopSystemUID

This is the highest UID on the current system that refers to a root UID. This is used to make sure that the temporary directory is owned by a system UID (`root`, `bin`, `sys` etc) rather than simply by root.

This is required since on many unix systems `/tmp` is not owned by root.

Default is to assume that any UID less than or equal to 10 is a root UID.

```
File::Temp->top_system_uid(10);  
my $topid = File::Temp->top_system_uid;
```

This value can be adjusted to reduce security checking if required. The value is only relevant when `safe_level` is set to `MEDIUM` or higher.

WARNING

For maximum security, endeavour always to avoid ever looking at, touching, or even imputing the existence of the filename. You do not know that that filename is connected to the same file as the handle you have, and attempts to check this can only trigger more race conditions. It's far more secure to use the filehandle alone and dispense with the filename altogether.

If you need to pass the handle to something that expects a filename then, on a unix system, use `"/dev/fd/" . fileno($fh)` for arbitrary programs, or more generally `< "+<=&" . fileno($fh)` for Perl programs. You will have to clear the close-on-exec bit on that file descriptor before passing it to another process.

```
use Fcntl qw/F_SETFD F_GETFD/;  
fcntl($tmpfh, F_SETFD, 0)  
    or die "Can't clear close-on-exec flag on temp fh: $!\n";
```

Temporary files and NFS

Some problems are associated with using temporary files that reside on NFS file systems and it is recommended that a local filesystem is used whenever possible. Some of the security tests will most probably fail when the temp file is not local. Additionally, be aware that the performance of I/O operations over NFS will not be as good as for a local disk.

HISTORY

Originally began life in May 1999 as an XS interface to the system `mkstemp()` function. In March 2000, the OpenBSD `mkstemp()` code was translated to Perl for total control of the code's security checking, to ensure the presence of the function regardless of operating system and to help with portability.

SEE ALSO

[*tmpnam*](#), [*tmpfile*](#), [*File::Spec*](#), [*File::Path*](#)

See [*IO::File*](#) and [*File::MkTemp*](#) for different implementations of temporary file handling.

AUTHOR

Tim Jenness <t.jenness@jach.hawaii.edu>

Copyright (C) 1999–2001 Tim Jenness and the UK Particle Physics and Astronomy Research Council. All Rights Reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Original Perl implementation loosely based on the OpenBSD C code for `mkstemp()`. Thanks to Tom Christiansen for suggesting that this module should be written and providing ideas for code improvements and security enhancements.

NAME

FileCache – keep more files open than the system permits

SYNOPSIS

```
use FileCache;
# or
use FileCache maxopen => 16;

cacheout $path;
print $path @data;

cacheout $mode, $path;
print $path @data;
```

DESCRIPTION

The `cacheout` function will make sure that there's a filehandle open for reading or writing available as the pathname you give it. It automatically closes and re-opens files if you exceed your system's maximum number of file descriptors, or the suggested maximum.

`cacheout` *EXPR*

The 1-argument form of `cacheout` will open a file for writing (`< ''`) on it's first use, and appending (`<< ''`) thereafter.

`cacheout` *MODE*, *EXPR*

The 2-argument form of `cacheout` will use the supplied mode for the initial and subsequent openings. Most valid modes for 3-argument `open` are supported namely; `< ''`, `< '+'`, `< '<'`, `< '<+'`, `<< ''`, `'|-'` and `'-|'`

CAVEATS

If you use `cacheout` with `'|-'` or `'-|'` you should catch `SIGPIPE` and explicitly close the filehandle., when it is closed from the other end some cleanup needs to be done.

While it is permissible to close a FileCache managed file, do not do so if you are calling `FileCache::cacheout` from a package other than which it was imported, or with another module which overrides `close`. If you must, use `FileCache::cacheout_close`.

BUGS

sys/param.h lies with its `NOFILE` define on some systems, so you may have to set `maxopen` (`$FileCache::cacheout_maxopen`) yourself.

NAME

FileHandle – supply object methods for filehandles

SYNOPSIS

```
use FileHandle;

$fh = new FileHandle;
if ($fh->open("< file")) {
    print <$fh>;
    $fh->close;
}

$fh = new FileHandle "> FOO";
if (defined $fh) {
    print $fh "bar\n";
    $fh->close;
}

$fh = new FileHandle "file", "r";
if (defined $fh) {
    print <$fh>;
    undef $fh;          # automatically closes the file
}

$fh = new FileHandle "file", O_WRONLY|O_APPEND;
if (defined $fh) {
    print $fh "corge\n";
    undef $fh;          # automatically closes the file
}

$pos = $fh->getpos;
$fh->setpos($pos);

$fh->setvbuf($buffer_var, _IOLBF, 1024);

($readfh, $writefh) = FileHandle::pipe;

autoflush STDOUT 1;
```

DESCRIPTION

NOTE: This class is now a front-end to the IO::* classes.

`FileHandle::new` creates a `FileHandle`, which is a reference to a newly created symbol (see the `Symbol` package). If it receives any parameters, they are passed to `FileHandle::open`; if the open fails, the `FileHandle` object is destroyed. Otherwise, it is returned to the caller.

`FileHandle::new_from_fd` creates a `FileHandle` like `new` does. It requires two parameters, which are passed to `FileHandle::fdopen`; if the `fdopen` fails, the `FileHandle` object is destroyed. Otherwise, it is returned to the caller.

`FileHandle::open` accepts one parameter or two. With one parameter, it is just a front end for the built-in `open` function. With two parameters, the first parameter is a filename that may include whitespace or other special characters, and the second parameter is the open mode, optionally followed by a file permission value.

If `FileHandle::open` receives a Perl mode string ("`"`", "`+`<", etc.) or a POSIX `fopen()` mode string ("`w`", "`r+`", etc.), it uses the basic Perl `open` operator.

If `FileHandle::open` is given a numeric mode, it passes that mode and the optional permissions value to the Perl `sysopen` operator. For convenience, `FileHandle::import` tries to import the `O_XXX` constants from the `Fcntl` module. If dynamic loading is not available, this may fail, but the rest of

FileHandle will still work.

`FileHandle::fdopen` is like `open` except that its first parameter is not a filename but rather a file handle name, a FileHandle object, or a file descriptor number.

If the C functions `fgetpos()` and `fsetpos()` are available, then `FileHandle::getpos` returns an opaque value that represents the current position of the FileHandle, and `FileHandle::setpos` uses that value to return to a previously visited position.

If the C function `setvbuf()` is available, then `FileHandle::setvbuf` sets the buffering policy for the FileHandle. The calling sequence for the Perl function is the same as its C counterpart, including the macros `_IOFBF`, `_IOLBF`, and `_IONBF`, except that the buffer parameter specifies a scalar variable to use as a buffer. **WARNING:** A variable used as a buffer by `FileHandle::setvbuf` must not be modified in any way until the FileHandle is closed or until `FileHandle::setvbuf` is called again, or memory corruption may result!

See [perlfunc](#) for complete descriptions of each of the following supported FileHandle methods, which are just front ends for the corresponding built-in functions:

```
close
fileno
getc
gets
eof
clearerr
seek
tell
```

See [perlvar](#) for complete descriptions of each of the following supported FileHandle methods:

```
autoflush
output_field_separator
output_record_separator
input_record_separator
input_line_number
format_page_number
format_lines_per_page
format_lines_left
format_name
format_top_name
format_line_break_characters
format_formfeed
```

Furthermore, for doing normal I/O you might need these:

`$fh->print`

See [print](#).

`$fh->printf`

See [printf](#).

`$fh->getline`

This works like `<$fh` described in [I/O Operators in perlop](#) except that it's more readable and can be safely called in a list context but still returns just one line.

`$fh->getlines`

This works like `<$fh` when called in a list context to read all the remaining lines in a file, except that it's more readable. It will also `croak()` if accidentally called in a scalar context.

There are many other functions available since FileHandle is descended from `IO::File`, `IO::Seekable`, and

IO::Handle. Please see those respective pages for documentation on more functions.

SEE ALSO

The **IO** extension, [perlfunc](#), *I/O Operators in perlop*.

NAME

filetest – Perl pragma to control the filetest permission operators

SYNOPSIS

```
$scan_perhaps_read = -r "file";      # use the mode bits
{
    use filetest 'access';           # intuit harder
    $scan_really_read = -r "file";
}
$scan_perhaps_read = -r "file";      # use the mode bits again
```

DESCRIPTION

This pragma tells the compiler to change the behaviour of the filetest permissions operators, the `-r -w -x -R -W -X` (see [perlfunc](#)).

The default behaviour to use the mode bits as returned by the `stat()` family of calls. This, however, may not be the right thing to do if for example various ACL (access control lists) schemes are in use. For such environments, `use filetest` may help the permission operators to return results more consistent with other tools.

Each "use filetest" or "no filetest" affects statements to the end of the enclosing block.

There may be a slight performance decrease in the filetests when `use filetest` is in effect, because in some systems the extended functionality needs to be emulated.

NOTE: using the file tests for security purposes is a lost cause from the start: there is a window open for race conditions (who is to say that the permissions will not change between the test and the real operation?). Therefore if you are serious about security, just try the real operation and test for its success. Think atomicity.

subpragma access

Currently only one subpragma, `access` is implemented. It enables (or disables) the use of `access()` or similar system calls. This extended filetest functionality is used only when the argument of the operators is a filename, not when it is a filehandle.

NAME

Filter::Simple – Simplified source filtering

SYNOPSIS

```
# in MyFilter.pm:

package MyFilter;

use Filter::Simple;

FILTER { ... };

# or just:
#
# use Filter::Simple sub { ... };

# in user's code:

use MyFilter;

# this code is filtered

no MyFilter;

# this code is not
```

DESCRIPTION**The Problem**

Source filtering is an immensely powerful feature of recent versions of Perl. It allows one to extend the language itself (e.g. the Switch module), to simplify the language (e.g. Language::Pythonesque), or to completely recast the language (e.g. Lingua::Romana::Perligata). Effectively, it allows one to use the full power of Perl as its own, recursively applied, macro language.

The excellent Filter::Util::Call module (by Paul Marquess) provides a usable Perl interface to source filtering, but it is often too powerful and not nearly as simple as it could be.

To use the module it is necessary to do the following:

1. Download, build, and install the Filter::Util::Call module. (If you have Perl 5.7.1 or later, this is already done for you.)
2. Set up a module that does a `use Filter::Util::Call`.
3. Within that module, create an `import` subroutine.
4. Within the `import` subroutine do a call to `filter_add`, passing it either a subroutine reference.
5. Within the subroutine reference, call `filter_read` or `filter_read_exact` to "prime" `$_` with source code data from the source file that will use your module. Check the status value returned to see if any source code was actually read in.
6. Process the contents of `$_` to change the source code in the desired manner.
7. Return the status value.
8. If the act of unimporting your module (via a `no`) should cause source code filtering to cease, create an `unimport` subroutine, and have it call `filter_del`. Make sure that the call to `filter_read` or `filter_read_exact` in step 5 will not accidentally read past the `no`. Effectively this limits source code filters to line-by-line operation, unless the `import` subroutine does some fancy pre-pre-parsing of the source code it's filtering.

For example, here is a minimal source code filter in a module named BANG.pm. It simply converts every occurrence of the sequence `BANG\s+BANG` to the sequence `die 'BANG' if $BANG` in any piece of code following a `use BANG;` statement (until the next `no BANG;` statement, if any):

```

package BANG;

use Filter::Util::Call ;

sub import {
    filter_add( sub {
        my $caller = caller;
        my ($status, $no_seen, $data);
        while ($status = filter_read()) {
            if (/^\s*no\s+$caller\s*;\s*?$/ ) {
                $no_seen=1;
                last;
            }
            $data .= $_;
            $_ = "";
        }
        $_ = $data;
        s/BANG\s+BANG/die 'BANG' if \xBANG/g
        unless $status < 0;
        $_ .= "no $class;\n" if $no_seen;
        return 1;
    })
}

sub unimport {
    filter_del();
}

1 ;

```

This level of sophistication puts filtering out of the reach of many programmers.

A Solution

The `Filter::Simple` module provides a simplified interface to `Filter::Util::Call`; one that is sufficient for most common cases.

Instead of the above process, with `Filter::Simple` the task of setting up a source code filter is reduced to:

1. Download and install the `Filter::Simple` module. (If you have Perl 5.7.1 or later, this is already done for you.)
2. Set up a module that does a `use Filter::Simple` and then calls `FILTER { ... }`.
3. Within the anonymous subroutine or block that is passed to `FILTER`, process the contents of `$_` to change the source code in the desired manner.

In other words, the previous example, would become:

```

package BANG;
use Filter::Simple;

FILTER {
    s/BANG\s+BANG/die 'BANG' if \xBANG/g;
};

1 ;

```

Note that the source code is passed as a single string, so any regex that uses `^` or `$` to detect line boundaries will need the `/m` flag.

Disabling or changing <no behaviour

By default, the installed filter only filters up to a line consisting of one of the three standard source "terminators":

```
no ModuleName;  # optional comment
```

or:

```
__END__
```

or:

```
__DATA__
```

but this can be altered by passing a second argument to use `Filter::Simple` or `FILTER` (just remember: there's *no* comma after the initial block when you use `FILTER`).

That second argument may be either a qr'd regular expression (which is then used to match the terminator line), or a defined false value (which indicates that no terminator line should be looked for), or a reference to a hash (in which case the terminator is the value associated with the key 'terminator').

For example, to cause the previous filter to filter only up to a line of the form:

```
GNAB esu;
```

you would write:

```
package BANG;
use Filter::Simple;

FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
}
qr/^\s*GNAB\s+esu\s*;\s*?$/;
```

or:

```
FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
}
{ terminator => qr/^\s*GNAB\s+esu\s*;\s*?$/ };
```

and to prevent the filter's being turned off in any way:

```
package BANG;
use Filter::Simple;

FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
}
";    # or: 0
```

or:

```
FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
}
{ terminator => "" };
```

Note that, no matter what you set the terminator pattern to, the actual terminator itself *must* be contained on a single source line.

All-in-one interface

Separating the loading of `Filter::Simple`:

```
use Filter::Simple;
```

from the setting up of the filtering:

```
FILTER { ... };
```

is useful because it allows other code (typically parser support code or caching variables) to be defined before the filter is invoked. However, there is often no need for such a separation.

In those cases, it is easier to just append the filtering subroutine and any terminator specification directly to the `use` statement that loads `Filter::Simple`, like so:

```
use Filter::Simple sub {
    s/BANG\s+BANG/die 'BANG' if \xBANG/g;
};
```

This is exactly the same as:

```
use Filter::Simple;
BEGIN {
    Filter::Simple::FILTER {
        s/BANG\s+BANG/die 'BANG' if \xBANG/g;
    };
}
```

except that the `FILTER` subroutine is not exported by `Filter::Simple`.

Filtering only specific components of source code

One of the problems with a filter like:

```
use Filter::Simple;

FILTER { s/BANG\s+BANG/die 'BANG' if \xBANG/g };
```

is that it indiscriminately applies the specified transformation to the entire text of your source program. So something like:

```
warn 'BANG BANG, YOU'RE DEAD';
BANG BANG;
```

will become:

```
warn 'die 'BANG' if $BANG, YOU'RE DEAD';
die 'BANG' if $BANG;
```

It is very common when filtering source to only want to apply the filter to the non-character-string parts of the code, or alternatively to *only* the character strings.

`Filter::Simple` supports this type of filtering by automatically exporting the `FILTER_ONLY` subroutine.

`FILTER_ONLY` takes a sequence of specifiers that install separate (and possibly multiple) filters that act on only parts of the source code. For example:

```
use Filter::Simple;

FILTER_ONLY
    code      => sub { s/BANG\s+BANG/die 'BANG' if \xBANG/g },
    quotelike => sub { s/BANG\s+BANG/CHITTY CHITTY/g };
```

The "code" subroutine will only be used to filter parts of the source code that are not quotelikes, POD, or `__DATA__`. The `quotelike` subroutine only filters Perl quotelikes (including here documents).

The full list of alternatives is:

"code"

Filters only those sections of the source code that are not quotelikes, POD, or `__DATA__`.

"executable"

Filters only those sections of the source code that are not POD or `__DATA__`.

"quotelike"

Filters only Perl quotelikes (as interpreted by `&Text::Balanced::extract_quotelike`).

"string"

Filters only the string literal parts of a Perl quotelike (i.e. the contents of a string literal, either half of a `tr///`, the second half of an `s///`).

"regex"

Filters only the pattern literal parts of a Perl quotelike (i.e. the contents of a `qr//` or an `m//`, the first half of an `s///`).

"all"

Filters everything. Identical in effect to `FILTER`.

Except for `< FILTER_ONLY code = sub { ... }`, each of the component filters is called repeatedly, once for each component found in the source code.

Note that you can also apply two or more of the same type of filter in a single `FILTER_ONLY`. For example, here's a simple macro-preprocessor that is only applied within regexes, with a final debugging pass that prints the resulting source code:

```
use Regexp::Common;
FILTER_ONLY
    regex => sub { s/!\[/[^\]/g },
    regex => sub { s/%d/$RE{num}{int}/g },
    regex => sub { s/%f/$RE{num}{real}/g },
    all   => sub { print if $::DEBUG };
```

Filtering only the code parts of source code

Most source code ceases to be grammatically correct when it is broken up into the pieces between string literals and regexes. So the 'code' component filter behaves slightly differently from the other partial filters described in the previous section.

Rather than calling the specified processor on each individual piece of code (i.e. on the bits between quotelikes), the 'code' partial filter operates on the entire source code, but with the quotelike bits "blanked out".

That is, a 'code' filter *replaces* each quoted string, quotelike, regex, POD, and `__DATA__` section with a placeholder. The delimiters of this placeholder are the contents of the `$;` variable at the time the filter is applied (normally `"\034"`). The remaining four bytes are a unique identifier for the component being replaced.

This approach makes it comparatively easy to write code preprocessors without worrying about the form or contents of strings, regexes, etc. For convenience, during a 'code' filtering operation, `Filter::Simple` provides a package variable (`$Filter::Simple::placeholder`) that contains a pre-compiled regex that matches any placeholder. Placeholders can be moved and re-ordered within the source code as needed.

Once the filtering has been applied, the original strings, regexes, POD, etc. are re-inserted into the code, by replacing each placeholder with the corresponding original component.

For example, the following filter detects concatenated pairs of strings/quotelikes and reverses the order in which they are concatenated:

```
package DemoRevCat;
use Filter::Simple;

FILTER_ONLY code => sub { my $ph = $Filter::Simple::placeholder;
                          s{ ($ph) \s* [.] \s* ($ph) }{ $2.$1 }gx
                        };

```

Thus, the following code:

```
use DemoRevCat;

my $str = "abc" . q(def);

print "$str\n";

```

would become:

```
my $str = q(def). "abc";

print "$str\n";

```

and hence print:

```
defabc

```

Using Filter::Simple with an explicit import subroutine

Filter::Simple generates a special import subroutine for your module (see ["How it works"](#)) which would normally replace any import subroutine you might have explicitly declared.

However, Filter::Simple is smart enough to notice your existing import and Do The Right Thing with it. That is, if you explicitly define an import subroutine in a package that's using Filter::Simple, that import subroutine will still be invoked immediately after any filter you install.

The only thing you have to remember is that the import subroutine *must* be declared *before* the filter is installed. If you use FILTER to install the filter:

```
package Filter::TurnItUpTo11;
use Filter::Simple;

FILTER { s/(\w+)/\U$1/ };

```

that will almost never be a problem, but if you install a filtering subroutine by passing it directly to the use Filter::Simple statement:

```
package Filter::TurnItUpTo11;
use Filter::Simple sub{ s/(\w+)/\U$1/ };

```

then you must make sure that your import subroutine appears before that use statement.

Using Filter::Simple and Exporter together

Likewise, Filter::Simple is also smart enough to Do The Right Thing if you use Exporter:

```
package Switch;
use base Exporter;
use Filter::Simple;

@EXPORT      = qw(switch case);
@EXPORT_OK   = qw(given when);

FILTER { $_ = magic_Perl_filter($_) }

```

Immediately after the filter has been applied to the source, Filter::Simple will pass control to Exporter, so it can do its magic too.

Of course, here too, Filter::Simple has to know you're using Exporter before it applies the filter. That's

almost never a problem, but if you're nervous about it, you can guarantee that things will work correctly by ensuring that your `use base Exporter` always precedes your `use Filter::Simple`.

How it works

The `Filter::Simple` module exports into the package that calls `FILTER` (or uses it directly) — such as package "BANG" in the above example — two automatically constructed subroutines — `import` and `unimport` — which take care of all the nasty details.

In addition, the generated `import` subroutine passes its own argument list to the filtering subroutine, so the BANG.pm filter could easily be made parametric:

```
package BANG;

use Filter::Simple;

FILTER {
    my ($die_msg, $var_name) = @_;
    s/BANG\s+BANG/die '$die_msg' if \${$var_name}/g;
};

# and in some user code:

use BANG "BOOM", "BAM"; # "BANG BANG" becomes: die 'BOOM' if $BAM
```

The specified filtering subroutine is called every time a `use BANG` is encountered, and passed all the source code following that call, up to either the next `no BANG;` (or whatever terminator you've set) or the end of the source file, whichever occurs first. By default, any `no BANG;` call must appear by itself on a separate line, or it is ignored.

AUTHOR

Damian Conway (damian@conway.org)

COPYRIGHT

Copyright (c) 2000–2001, Damian Conway. All Rights Reserved.
This module is free software. It may be used, redistributed
and/or modified under the same terms as Perl itself.

NAME

Filter::Util::Call – Perl Source Filter Utility Module

SYNOPSIS

```
use Filter::Util::Call ;
```

DESCRIPTION

This module provides you with the framework to write *Source Filters* in Perl.

An alternate interface to Filter::Util::Call is now available. See [Filter::Simple](#) for more details.

A *Perl Source Filter* is implemented as a Perl module. The structure of the module can take one of two broadly similar formats. To distinguish between them, the first will be referred to as *method filter* and the second as *closure filter*.

Here is a skeleton for the *method filter*:

```
package MyFilter ;
use Filter::Util::Call ;
sub import
{
    my($type, @arguments) = @_ ;
    filter_add([]) ;
}
sub filter
{
    my($self) = @_ ;
    my($status) ;

    $status = filter_read() ;
    $status ;
}
1 ;
```

and this is the equivalent skeleton for the *closure filter*:

```
package MyFilter ;
use Filter::Util::Call ;
sub import
{
    my($type, @arguments) = @_ ;
    filter_add(
        sub
        {
            my($status) ;
            $status = filter_read() ;
            $status ;
        } )
}
1 ;
```

To make use of either of the two filter modules above, place the line below in a Perl source file.

```
use MyFilter;
```

In fact, the skeleton modules shown above are fully functional *Source Filters*, albeit fairly useless ones. All they do is filter the source stream without modifying it at all.

As you can see both modules have a broadly similar structure. They both make use of the `Filter::Util::Call` module and both have an `import` method. The difference between them is that the *method filter* requires a *filter* method, whereas the *closure filter* gets the equivalent of a *filter* method with the anonymous sub passed to *filter_add*.

To make proper use of the *closure filter* shown above you need to have a good understanding of the concept of a *closure*. See [perlref](#) for more details on the mechanics of *closures*.

use Filter::Util::Call

The following functions are exported by `Filter::Util::Call`:

```
filter_add()
filter_read()
filter_read_exact()
filter_del()
```

import()

The `import` method is used to create an instance of the filter. It is called indirectly by Perl when it encounters the `use MyFilter` line in a source file (See [import](#) for more details on `import`).

It will always have at least one parameter automatically passed by Perl – this corresponds to the name of the package. In the example above it will be `"MyFilter"`.

Apart from the first parameter, `import` can accept an optional list of parameters. These can be used to pass parameters to the filter. For example:

```
use MyFilter qw(a b c) ;
```

will result in the `@_` array having the following values:

```
@_ [0] => "MyFilter"
@_ [1] => "a"
@_ [2] => "b"
@_ [3] => "c"
```

Before terminating, the `import` function must explicitly install the filter by calling *filter_add*.

filter_add()

The function, *filter_add*, actually installs the filter. It takes one parameter which should be a reference. The kind of reference used will dictate which of the two filter types will be used.

If a CODE reference is used then a *closure filter* will be assumed.

If a CODE reference is not used, a *method filter* will be assumed. In a *method filter*, the reference can be used to store context information. The reference will be *blessed* into the package by *filter_add*.

See the filters at the end of this document for examples of using context information using both *method filters* and *closure filters*.

filter() and anonymous sub

Both the *filter* method used with a *method filter* and the anonymous sub used with a *closure filter* is where the main processing for the filter is done.

The big difference between the two types of filter is that the *method filter* uses the object passed to the method to store any context data, whereas the *closure filter* uses the lexical variables that are maintained by the closure.

Note that the single parameter passed to the *method filter*, `$self`, is the same reference that was passed to *filter_add* blessed into the filter's package. See the example filters later on for details of using `$self`.

Here is a list of the common features of the anonymous sub and the `filter()` method.

\$_ Although `$_` doesn't actually appear explicitly in the sample filters above, it is implicitly used in a number of places.

Firstly, when either `filter` or the anonymous sub are called, a local copy of `$_` will automatically be created. It will always contain the empty string at this point.

Next, both `filter_read` and `filter_read_exact` will append any source data that is read to the end of `$_`.

Finally, when `filter` or the anonymous sub are finished processing, they are expected to return the filtered source using `$_`.

This implicit use of `$_` greatly simplifies the filter.

\$status

The status value that is returned by the user's `filter` method or anonymous sub and the `filter_read` and `read_exact` functions take the same set of values, namely:

```
< 0  Error
= 0  EOF
> 0  OK
```

filter_read and filter_read_exact

These functions are used by the filter to obtain either a line or block from the next filter in the chain or the actual source file if there aren't any other filters.

The function `filter_read` takes two forms:

```
$status = filter_read() ;
$status = filter_read($size) ;
```

The first form is used to request a *line*, the second requests a *block*.

In line mode, `filter_read` will append the next source line to the end of the `$_` scalar.

In block mode, `filter_read` will append a block of data which is `<= $size` to the end of the `$_` scalar. It is important to emphasise that `filter_read` will not necessarily read a block which is *precisely* `$size` bytes.

If you need to be able to read a block which has an exact size, you can use the function `filter_read_exact`. It works identically to `filter_read` in block mode, except it will try to read a block which is exactly `$size` bytes in length. The only circumstances when it will not return a block which is `$size` bytes long is on EOF or error.

It is *very* important to check the value of `$status` after *every* call to `filter_read` or `filter_read_exact`.

filter_del

The function, `filter_del`, is used to disable the current filter. It does not affect the running of the filter. All it does is tell Perl not to call filter any more.

See [Example 4: Using filter_del](#) for details.

EXAMPLES

Here are a few examples which illustrate the key concepts – as such most of them are of little practical use.

The `examples` sub-directory has copies of all these filters implemented both as *method filters* and as *closure filters*.

Example 1: A simple filter.

Below is a *method filter* which is hard-wired to replace all occurrences of the string "Joe" to "Jim". Not particularly Useful, but it is the first example and I wanted to keep it simple.

```
package Joe2Jim ;
use Filter::Util::Call ;
sub import
{
    my($type) = @_ ;
    filter_add(bless []) ;
}
sub filter
{
    my($self) = @_ ;
    my($status) ;
    s/Joe/Jim/g
    if ($status = filter_read()) > 0 ;
    $status ;
}
1 ;
```

Here is an example of using the filter:

```
use Joe2Jim ;
print "Where is Joe?\n" ;
```

And this is what the script above will print:

```
Where is Jim?
```

Example 2: Using the context

The previous example was not particularly useful. To make it more general purpose we will make use of the context data and allow any arbitrary *from* and *to* strings to be used. This time we will use a *closure filter*. To reflect its enhanced role, the filter is called Subst.

```
package Subst ;
use Filter::Util::Call ;
use Carp ;
sub import
{
    croak("usage: use Subst qw(from to)")
    unless @_ == 3 ;
    my ($self, $from, $to) = @_ ;
    filter_add(
        sub
        {
            my ($status) ;
            s/$from/$to/
            if ($status = filter_read()) > 0 ;
            $status ;
        })
}
1 ;
```

and is used like this:

```
use Subst qw(Joe Jim) ;
print "Where is Joe?\n" ;
```

Example 3: Using the context within the filter

Here is a filter which a variation of the `Joe2Jim` filter. As well as substituting all occurrences of "Joe" to "Jim" it keeps a count of the number of substitutions made in the context object.

Once EOF is detected (`$status` is zero) the filter will insert an extra line into the source stream. When this extra line is executed it will print a count of the number of substitutions actually made. Note that `$status` is set to 1 in this case.

```
package Count ;
use Filter::Util::Call ;
sub filter
{
    my ($self) = @_ ;
    my ($status) ;

    if (($status = filter_read()) > 0 ) {
        s/Joe/Jim/g ;
        ++ $$self ;
    }
    elsif ($$self >= 0) { # EOF
        $_ = "print q[Made ${$self} substitutions\n]" ;
        $status = 1 ;
        $$self = -1 ;
    }

    $status ;
}

sub import
{
    my ($self) = @_ ;
    my ($count) = 0 ;
    filter_add(\$count) ;
}

1 ;
```

Here is a script which uses it:

```
use Count ;
print "Hello Joe\n" ;
print "Where is Joe\n" ;
```

Outputs:

```
Hello Jim
Where is Jim
Made 2 substitutions
```

Example 4: Using `filter_del`

Another variation on a theme. This time we will modify the `Subst` filter to allow a starting and stopping pattern to be specified as well as the *from* and *to* patterns. If you know the *vi* editor, it is the equivalent of this command:

```
:/start/,/stop/s/from/to/
```


When used as a filter we want to invoke it like this:

```
use NewSubst qw(start stop from to) ;
```

Here is the module.

```
package NewSubst ;
use Filter::Util::Call ;
use Carp ;
sub import
{
    my ($self, $start, $stop, $from, $to) = @_ ;
    my ($found) = 0 ;
    croak("usage: use Subst qw(start stop from to)")
        unless @_ == 5 ;

    filter_add(
        sub
        {
            my ($status) ;
            if (($status = filter_read()) > 0) {
                $found = 1
                if $found == 0 and /$start/ ;

                if ($found) {
                    s/$from/$to/ ;
                    filter_del() if /$stop/ ;
                }
            }
            $status ;
        } )
    }
    1 ;
}
```

Filter::Simple

If you intend using the Filter::Call functionality, I would strongly recommend that you check out Damian Conway's excellent Filter::Simple module. This module provides a much cleaner interface than Filter::Util::Call. Although it doesn't allow the fine control that Filter::Util::Call does, it should be adequate for the majority of applications. It's available at

http://www.cpan.org/modules/by-author/Damian_Conway/Filter-Simple.tar.gz
<http://www.csse.monash.edu.au/~damian/CPAN/Filter-Simple.tar.gz>

AUTHOR

Paul Marquess

DATE

26th January 1996

NAME

FindBin – Locate directory of original perl script

SYNOPSIS

```
use FindBin;
use lib "$FindBin::Bin/./lib";

or

use FindBin qw($Bin);
use lib "$Bin/./lib";
```

DESCRIPTION

Locates the full path to the script bin directory to allow the use of paths relative to the bin directory.

This allows a user to setup a directory tree for some software with directories <root>/bin and <root>/lib and then the above example will allow the use of modules in the lib directory without knowing where the software tree is installed.

If perl is invoked using the `-e` option or the perl script is read from STDIN then FindBin sets both `$Bin` and `$RealBin` to the current directory.

EXPORTABLE VARIABLES

<code>\$Bin</code>	- path to bin directory from where script was invoked
<code>\$Script</code>	- basename of script from which perl was invoked
<code>\$RealBin</code>	- <code>\$Bin</code> with all links resolved
<code>\$RealScript</code>	- <code>\$Script</code> with all links resolved

KNOWN ISSUES

If there are two modules using FindBin from different directories under the same interpreter, this won't work. Since FindBin uses BEGIN block, it'll be executed only once, and only the first caller will get it right. This is a problem under mod_perl and other persistent Perl environments, where you shouldn't use this module. Which also means that you should avoid using FindBin in modules that you plan to put on CPAN. The only way to make sure that FindBin will work is to force the BEGIN block to be executed again:

```
delete $INC{'FindBin.pm'};
require FindBin;
```

KNOWN BUGS

If perl is invoked as

```
perl filename
```

and *filename* does not have executable rights and a program called *filename* exists in the users `$ENV{PATH}` which satisfies both `-x` and `-T` then FindBin assumes that it was invoked via the `$ENV{PATH}`.

Workaround is to invoke perl as

```
perl ./filename
```

AUTHORS

FindBin is supported as part of the core perl distribution. Please send bug reports to [<perlbug@perl.org>](mailto:perlbug@perl.org) using the perlbug program included with perl.

Graham Barr [<gbarr@pobox.com>](mailto:gbarr@pobox.com) Nick Ing-Simmons [<nik@tiuk.ti.com>](mailto:nik@tiuk.ti.com)

COPYRIGHT

Copyright (c) 1995 Graham Barr & Nick Ing-Simmons. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

Getopt::Long – Extended processing of command line options

SYNOPSIS

```
use Getopt::Long;
my $data    = "file.dat";
my $length  = 24;
my $verbose;
$result = GetOptions ("length=i" => \$length,    # numeric
                     "file=s"   => \$data,       # string
                     "verbose"  => \$verbose);    # flag
```

DESCRIPTION

The Getopt::Long module implements an extended getopt function called `GetOptions()`. This function adheres to the POSIX syntax for command line options, with GNU extensions. In general, this means that options have long names instead of single letters, and are introduced with a double dash "`--`". Support for bundling of command line options, as was the case with the more traditional single-letter approach, is provided but not enabled by default.

Command Line Options, an Introduction

Command line operated programs traditionally take their arguments from the command line, for example filenames or other information that the program needs to know. Besides arguments, these programs often take command line *options* as well. Options are not necessary for the program to work, hence the name 'option', but are used to modify its default behaviour. For example, a program could do its job quietly, but with a suitable option it could provide verbose information about what it did.

Command line options come in several flavours. Historically, they are preceded by a single dash `-`, and consist of a single letter.

```
-l -a -c
```

Usually, these single-character options can be bundled:

```
-lac
```

Options can have values, the value is placed after the option character. Sometimes with whitespace in between, sometimes not:

```
-s 24 -s24
```

Due to the very cryptic nature of these options, another style was developed that used long names. So instead of a cryptic `-l` one could use the more descriptive `--long`. To distinguish between a bundle of single-character options and a long one, two dashes are used to precede the option name. Early implementations of long options used a plus `+` instead. Also, option values could be specified either like

```
--size=24
```

or

```
--size 24
```

The `+` form is now obsolete and strongly deprecated.

Getting Started with Getopt::Long

Getopt::Long is the Perl5 successor of `newgetopt.pl`. This was the first Perl module that provided support for handling the new style of command line options, hence the name Getopt::Long. This module also supports single-character options and bundling. In this case, the options are restricted to alphabetic characters only, and the characters `?` and `-`.

To use Getopt::Long from a Perl program, you must include the following line in your Perl program:

```
use Getopt::Long;
```

This will load the core of the `Getopt::Long` module and prepare your program for using it. Most of the actual `Getopt::Long` code is not loaded until you really call one of its functions.

In the default configuration, options names may be abbreviated to uniqueness, case does not matter, and a single dash is sufficient, even for long option names. Also, options may be placed between non-option arguments. See [Configuring Getopt::Long](#) for more details on how to configure `Getopt::Long`.

Simple options

The most simple options are the ones that take no values. Their mere presence on the command line enables the option. Popular examples are:

```
--all --verbose --quiet --debug
```

Handling simple options is straightforward:

```
my $verbose = ''; # option variable with default value (false)
my $all = '';    # option variable with default value (false)
GetOptions ('verbose' => \$verbose, 'all' => \$all);
```

The call to `GetOptions()` parses the command line arguments that are present in `@ARGV` and sets the option variable to the value 1 if the option did occur on the command line. Otherwise, the option variable is not touched. Setting the option value to true is often called *enabling* the option.

The option name as specified to the `GetOptions()` function is called the option *specification*. Later we'll see that this specification can contain more than just the option name. The reference to the variable is called the option *destination*.

`GetOptions()` will return a true value if the command line could be processed successfully. Otherwise, it will write error messages to `STDERR`, and return a false result.

A little bit less simple options

`Getopt::Long` supports two useful variants of simple options: *negatable* options and *incremental* options.

A negatable option is specified with an exclamation mark `!` after the option name:

```
my $verbose = ''; # option variable with default value (false)
GetOptions ('verbose!' => \$verbose);
```

Now, using `--verbose` on the command line will enable `$verbose`, as expected. But it is also allowed to use `--noverbose`, which will disable `$verbose` by setting its value to `.` Using a suitable default value, the program can find out whether `$verbose` is false by default, or disabled by using `--noverbose`.

An incremental option is specified with a plus `+` after the option name:

```
my $verbose = ''; # option variable with default value (false)
GetOptions ('verbose+' => \$verbose);
```

Using `--verbose` on the command line will increment the value of `$verbose`. This way the program can keep track of how many times the option occurred on the command line. For example, each occurrence of `--verbose` could increase the verbosity level of the program.

Mixing command line option with other arguments

Usually programs take command line options as well as other arguments, for example, file names. It is good practice to always specify the options first, and the other arguments last. `Getopt::Long` will, however, allow the options and arguments to be mixed and 'filter out' all the options before passing the rest of the arguments to the program. To stop `Getopt::Long` from processing further arguments, insert a double dash `--` on the command line:

```
--size 24 -- --all
```

In this example, `--all` will *not* be treated as an option, but passed to the program unharmed, in `@ARGV`.

Options with values

For options that take values it must be specified whether the option value is required or not, and what kind of value the option expects.

Three kinds of values are supported: integer numbers, floating point numbers, and strings.

If the option value is required, `Getopt::Long` will take the command line argument that follows the option and assign this to the option variable. If, however, the option value is specified as optional, this will only be done if that value does not look like a valid command line option itself.

```
my $tag = '';          # option variable with default value
GetOptions ('tag=s' => \$tag);
```

In the option specification, the option name is followed by an equals sign `=` and the letter `s`. The equals sign indicates that this option requires a value. The letter `s` indicates that this value is an arbitrary string. Other possible value types are `i` for integer values, and `f` for floating point values. Using a colon `:` instead of the equals sign indicates that the option value is optional. In this case, if no suitable value is supplied, string valued options get an empty string `''` assigned, while numeric options are set to `.`

Options with multiple values

Options sometimes take several values. For example, a program could use multiple directories to search for library files:

```
--library lib/stdlib --library lib/extlib
```

To accomplish this behaviour, simply specify an array reference as the destination for the option:

```
my @libfiles = ();
GetOptions ("library=s" => \@libfiles);
```

Used with the example above, `@libfiles` would contain two strings upon completion: `"lib/stdlib"` and `"lib/extlib"`, in that order. It is also possible to specify that only integer or floating point numbers are acceptable values.

Often it is useful to allow comma-separated lists of values as well as multiple occurrences of the options. This is easy using Perl's `split()` and `join()` operators:

```
my @libfiles = ();
GetOptions ("library=s" => \@libfiles);
@libfiles = split(/,/ , join(',', @libfiles));
```

Of course, it is important to choose the right separator string for each purpose.

Options with hash values

If the option destination is a reference to a hash, the option will take, as value, strings of the form *key=value*. The value will be stored with the specified key in the hash.

```
my %defines = ();
GetOptions ("define=s" => \%defines);
```

When used with command line options:

```
--define os=linux --define vendor=redhat
```

the hash `%defines` will contain two keys, `"os"` with value `"linux"` and `"vendor"` with value `"redhat"`. It is also possible to specify that only integer or floating point numbers are acceptable values. The keys are always taken to be strings.

User-defined subroutines to handle options

Ultimate control over what should be done when (actually: each time) an option is encountered on the command line can be achieved by designating a reference to a subroutine (or an anonymous subroutine) as the option destination. When `GetOptions()` encounters the option, it will call the subroutine with two or

three arguments. The first argument is the name of the option. For a scalar or array destination, the second argument is the value to be stored. For a hash destination, the second arguments is the key to the hash, and the third argument the value to be stored. It is up to the subroutine to store the value, or do whatever it thinks is appropriate.

A trivial application of this mechanism is to implement options that are related to each other. For example:

```
my $verbose = ''; # option variable with default value (false)
GetOptions ('verbose' => \$verbose,
           'quiet'    => sub { $verbose = 0 });
```

Here `--verbose` and `--quiet` control the same variable `$verbose`, but with opposite values.

If the subroutine needs to signal an error, it should call `die()` with the desired error message as its argument. `GetOptions()` will catch the `die()`, issue the error message, and record that an error result must be returned upon completion.

If the text of the error message starts with an exclamation mark `!` it is interpreted specially by `GetOptions()`. There is currently one special command implemented: `die("!FINISH")` will cause `GetOptions()` to stop processing options, as if it encountered a double dash `--`.

Options with multiple names

Often it is user friendly to supply alternate mnemonic names for options. For example `--height` could be an alternate name for `--length`. Alternate names can be included in the option specification, separated by vertical bar `|` characters. To implement the above example:

```
GetOptions ('length|height=f' => \$length);
```

The first name is called the *primary* name, the other names are called *aliases*.

Multiple alternate names are possible.

Case and abbreviations

Without additional configuration, `GetOptions()` will ignore the case of option names, and allow the options to be abbreviated to uniqueness.

```
GetOptions ('length|height=f' => \$length, "head" => \$head);
```

This call will allow `-l` and `-L` for the length option, but requires a least `-hea` and `-hei` for the head and height options.

Summary of Option Specifications

Each option specifier consists of two parts: the name specification and the argument specification.

The name specification contains the name of the option, optionally followed by a list of alternative names separated by vertical bar characters.

```
length          option name is "length"
length|size|l   name is "length", aliases are "size" and "l"
```

The argument specification is optional. If omitted, the option is considered boolean, a value of 1 will be assigned when the option is used on the command line.

The argument specification can be

- ! The option does not take an argument and may be negated, i.e. prefixed by "no". E.g. `"foo!"` will allow `--foo` (a value of 1 will be assigned) and `--nofoo` (a value of 0 will be assigned). If the option has aliases, this applies to the aliases as well.

Using negation on a single letter option when bundling is in effect is pointless and will result in a warning.

- + The option does not take an argument and will be incremented by 1 every time it appears on the command line. E.g. "more+", when used with `--more --more --more`, will increment the value three times, resulting in a value of 3 (provided it was 0 or undefined at first).

The + specifier is ignored if the option destination is not a scalar.

`= type [desttype]`

The option requires an argument of the given type. Supported types are:

- s String. An arbitrary sequence of characters. It is valid for the argument to start with `-` or `--`.
- i Integer. An optional leading plus or minus sign, followed by a sequence of digits.
- o Extended integer, Perl style. This can be either an optional leading plus or minus sign, followed by a sequence of digits, or an octal string (a zero, optionally followed by '0', '1', .. '7'), or a hexadecimal string (0x followed by '0' .. '9', 'a' .. 'f', case insensitive), or a binary string (0b followed by a series of '0' and '1').
- f Real number. For example 3.14, -6.23E24 and so on.

The *desttype* can be @ or % to specify that the option is list or a hash valued. This is only needed when the destination for the option value is not otherwise specified. It should be omitted when not needed.

`: type [desttype]`

Like `=`, but designates the argument as optional. If omitted, an empty string will be assigned to string values options, and the value zero to numeric options.

Note that if a string argument starts with `-` or `--`, it will be considered an option on itself.

`: number [desttype]`

Like `:i`, but if the value is omitted, the *number* will be assigned.

`: + [desttype]`

Like `:i`, but if the value is omitted, the current value for the option will be incremented.

Advanced Possibilities

Object oriented interface

Getopt::Long can be used in an object oriented way as well:

```
use Getopt::Long;
$p = new Getopt::Long::Parser;
$p->configure(...configuration options...);
if ($p->getoptions(...options descriptions...)) ...
```

Configuration options can be passed to the constructor:

```
$p = new Getopt::Long::Parser
    config => [...configuration options...];
```

Thread Safety

Getopt::Long is thread safe when using `ithreads` as of Perl 5.8. It is *not* thread safe when using the older (experimental and now obsolete) `threads` implementation that was added to Perl 5.005.

Documentation and help texts

Getopt::Long encourages the use of Pod::Usage to produce help messages. For example:

```
use Getopt::Long;
use Pod::Usage;

my $man = 0;
my $help = 0;
```

```

GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
pod2usage(1) if $help;
pod2usage(-exitstatus => 0, -verbose => 2) if $man;

__END__

=head1 NAME

sample - Using GetOpt::Long and Pod::Usage

=head1 SYNOPSIS

sample [options] [file ...]

    Options:
        -help             brief help message
        -man              full documentation

=head1 OPTIONS

=over 8

=item B<-help>

Print a brief help message and exits.

=item B<-man>

Prints the manual page and exits.

=back

=head1 DESCRIPTION

B<This program> will read the given input file(s) and do something
useful with the contents thereof.

=cut

```

See [Pod::Usage](#) for details.

Storing options in a hash

Sometimes, for example when there are a lot of options, having a separate variable for each of them can be cumbersome. `GetOptions()` supports, as an alternative mechanism, storing options in a hash.

To obtain this, a reference to a hash must be passed *as the first argument* to `GetOptions()`. For each option that is specified on the command line, the option value will be stored in the hash with the option name as key. Options that are not actually used on the command line will not be put in the hash, on other words, `exists($h{option})` (or `defined()`) can be used to test if an option was used. The drawback is that warnings will be issued if the program runs under `use strict` and uses `$h{option}` without testing with `exists()` or `defined()` first.

```

my %h = ();
GetOptions (\%h, 'length=i');      # will store in $h{length}

```

For options that take list or hash values, it is necessary to indicate this by appending an `@` or `%` sign after the type:

```

GetOptions (\%h, 'colours=s@');    # will push to @{$h{colours}}

```

To make things more complicated, the hash may contain references to the actual destinations, for example:

```

my $len = 0;
my %h = ('length' => \$len);
GetOptions (\%h, 'length=i');      # will store in $len

```


This example is fully equivalent with:

```
my $len = 0;
GetOptions ('length=i' => \$len); # will store in $len
```

Any mixture is possible. For example, the most frequently used options could be stored in variables while all other options get stored in the hash:

```
my $verbose = 0; # frequently referred
my $debug = 0; # frequently referred
my %h = ('verbose' => \$verbose, 'debug' => \$debug);
GetOptions (\%h, 'verbose', 'debug', 'filter', 'size=i');
if ( $verbose ) { ... }
if ( exists $h{filter} ) { ... option 'filter' was specified ... }
```

Bundling

With bundling it is possible to set several single-character options at once. For example if a, v and x are all valid options,

```
-vax
```

would set all three.

Getopt::Long supports two levels of bundling. To enable bundling, a call to Getopt::Long::Configure is required.

The first level of bundling can be enabled with:

```
Getopt::Long::Configure ("bundling");
```

Configured this way, single-character options can be bundled but long options **must** always start with a double dash – to avoid ambiguity. For example, when vax, a, v and x are all valid options,

```
-vax
```

would set a, v and x, but

```
--vax
```

would set vax.

The second level of bundling lifts this restriction. It can be enabled with:

```
Getopt::Long::Configure ("bundling_override");
```

Now, -vax would set the option vax.

When any level of bundling is enabled, option values may be inserted in the bundle. For example:

```
-h24w80
```

is equivalent to

```
-h 24 -w 80
```

When configured for bundling, single-character options are matched case sensitive while long options are matched case insensitive. To have the single-character options matched case insensitive as well, use:

```
Getopt::Long::Configure ("bundling", "ignorecase_always");
```

It goes without saying that bundling can be quite confusing.

The lonesome dash

Normally, a lone dash – on the command line will not be considered an option. Option processing will terminate (unless "permute" is configured) and the dash will be left in @ARGV.

It is possible to get special treatment for a lone dash. This can be achieved by adding an option specification

with an empty name, for example:

```
GetOptions ('' => \$stdio);
```

A lone dash on the command line will now be a legal option, and using it will set variable `$stdio`.

Argument callback

A special option `'name' <` can be used to designate a subroutine to handle non-option arguments. When `GetOptions()` encounters an argument that does not look like an option, it will immediately call this subroutine and passes it one parameter: the argument name.

For example:

```
my $width = 80;
sub process { ... }
GetOptions ('width=i' => \$width, '<>' => \&process);
```

When applied to the following command line:

```
arg1 --width=72 arg2 --width=60 arg3
```

This will call `process("arg1")` while `$width` is 80, `process("arg2")` while `$width` is 72, and `process("arg3")` while `$width` is 60.

This feature requires configuration option **permute**, see section [Configuring Getopt::Long](#).

Configuring Getopt::Long

`Getopt::Long` can be configured by calling subroutine `Getopt::Long::Configure()`. This subroutine takes a list of quoted strings, each specifying a configuration option to be enabled, e.g. `ignore_case`, or disabled, e.g. `no_ignore_case`. Case does not matter. Multiple calls to `Configure()` are possible.

Alternatively, as of version 2.24, the configuration options may be passed together with the `use` statement:

```
use Getopt::Long qw(:config no_ignore_case bundling);
```

The following options are available:

default	This option causes all configuration options to be reset to their default values.
posix_default	This option causes all configuration options to be reset to their default values as if the environment variable <code>POSIXLY_CORRECT</code> had been set.
auto_abbrev	Allow option names to be abbreviated to uniqueness. Default is enabled unless environment variable <code>POSIXLY_CORRECT</code> has been set, in which case <code>auto_abbrev</code> is disabled.
getopt_compat	Allow <code>+</code> to start options. Default is enabled unless environment variable <code>POSIXLY_CORRECT</code> has been set, in which case <code>getopt_compat</code> is disabled.
gnu_compat	<code>gnu_compat</code> controls whether <code>--opt=</code> is allowed, and what it should do. Without <code>gnu_compat</code> , <code>--opt=</code> gives an error. With <code>gnu_compat</code> , <code>--opt=</code> will give option <code>opt</code> and empty value. This is the way GNU <code>getopt_long()</code> does it.
gnu_getopt	This is a short way of setting <code>gnu_compat bundling permute no_getopt_compat</code> . With <code>gnu_getopt</code> , command line handling should be fully compatible with GNU <code>getopt_long()</code> .
require_order	Whether command line arguments are allowed to be mixed with options. Default is disabled unless environment variable <code>POSIXLY_CORRECT</code> has been set, in which case <code>require_order</code> is enabled.

See also `permute`, which is the opposite of `require_order`.

permute Whether command line arguments are allowed to be mixed with options. Default is enabled unless environment variable `POSIXLY_CORRECT` has been set, in which case `permute` is disabled. Note that `permute` is the opposite of `require_order`.

If `permute` is enabled, this means that

```
--foo arg1 --bar arg2 arg3
```

is equivalent to

```
--foo --bar arg1 arg2 arg3
```

If an argument callback routine is specified, `@ARGV` will always be empty upon successful return of `GetOptions()` since all options have been processed. The only exception is when `-` is used:

```
--foo arg1 --bar arg2 -- arg3
```

This will call the callback routine for `arg1` and `arg2`, and then terminate `GetOptions()` leaving "arg2" in `@ARGV`.

If `require_order` is enabled, options processing terminates when the first non-option is encountered.

```
--foo arg1 --bar arg2 arg3
```

is equivalent to

```
--foo -- arg1 --bar arg2 arg3
```

If `pass_through` is also enabled, options processing will terminate at the first unrecognized option, or non-option, whichever comes first.

bundling (default: disabled)

Enabling this option will allow single-character options to be bundled. To distinguish bundles from long option names, long options *must* be introduced with `-` and bundles with `.`.

Note that, if you have options `a`, `l` and `all`, and `auto_abbrev` enabled, possible arguments and option settings are:

using argument	sets option(s)
-----	-----
-a, --a	a
-l, --l	l
-al, -la, -ala, -all, ...	a, l
--al, --all	all

The suprising part is that `-a` sets option `a` (due to auto completion), not `all`.

Note: disabling `bundling` also disables `bundling_override`.

bundling_override (default: disabled)

If `bundling_override` is enabled, `bundling` is enabled as with `bundling` but now long option names override option bundles.

Note: disabling `bundling_override` also disables `bundling`.

Note: Using option bundling can easily lead to unexpected results, especially when mixing long options and bundles. Caveat emptor.

ignore_case (default: enabled)

If enabled, case is ignored when matching long option names. If, however, `bundling` is enabled as well, single character options will be treated case-sensitive.

With `ignore_case`, option specifications for options that only differ in case, e.g., `"foo"` and `"Foo"`, will be flagged as duplicates.

Note: disabling `ignore_case` also disables `ignore_case_always`.

`ignore_case_always` (default: disabled)

When bundling is in effect, case is ignored on single-character options also.

Note: disabling `ignore_case_always` also disables `ignore_case`.

`pass_through` (default: disabled)

Options that are unknown, ambiguous or supplied with an invalid option value are passed through in `@ARGV` instead of being flagged as errors. This makes it possible to write wrapper scripts that process only part of the user supplied command line arguments, and pass the remaining options to some other program.

If `require_order` is enabled, options processing will terminate at the first unrecognized option, or non-option, whichever comes first. However, if `permute` is enabled instead, results can become confusing.

`prefix` The string that starts options. If a constant string is not sufficient, see `prefix_pattern`.

`prefix_pattern` A Perl pattern that identifies the strings that introduce options. Default is `(-|\-|\+)` unless environment variable `POSIXLY_CORRECT` has been set, in which case it is `(-|-)`.

`debug` (default: disabled)

Enable debugging output.

Return values and Errors

Configuration errors and errors in the option definitions are signalled using `die()` and will terminate the calling program unless the call to `Getopt::Long::GetOptions()` was embedded in `eval { ... }`, or `die()` was trapped using `$SIG{__DIE__}`.

`GetOptions` returns true to indicate success. It returns false when the function detected one or more errors during option parsing. These errors are signalled using `warn()` and can be trapped with `$SIG{__WARN__}`.

Errors that can't happen are signalled using `Carp::croak()`.

Legacy

The earliest development of `newgetopt.pl` started in 1990, with Perl version 4. As a result, its development, and the development of `Getopt::Long`, has gone through several stages. Since backward compatibility has always been extremely important, the current version of `Getopt::Long` still supports a lot of constructs that nowadays are no longer necessary or otherwise unwanted. This section describes briefly some of these 'features'.

Default destinations

When no destination is specified for an option, `GetOptions` will store the resultant value in a global variable named `opt_XXX`, where `XXX` is the primary name of this option. When a program executes under `use strict` (recommended), these variables must be pre-declared with `our()` or `use vars`.

```
our $opt_length = 0;
GetOptions ('length=i');    # will store in $opt_length
```

To yield a usable Perl variable, characters that are not part of the syntax for variables are translated to underscores. For example, `-fpp-struct-return` will set the variable `$opt_fpp_struct_return`. Note that this variable resides in the namespace of the calling program, not necessarily `main`. For example:

```
GetOptions ("size=i", "sizes=i@");
```

with command line "`--size 10 --sizes 24 --sizes 48`" will perform the equivalent of the assignments

```
$opt_size = 10;
@opt_sizes = (24, 48);
```

Alternative option starters

A string of alternative option starter characters may be passed as the first argument (or the first argument after a leading hash reference argument).

```
my $len = 0;
GetOptions ('/', 'length=i' => $len);
```

Now the command line may look like:

```
/length 24 -- arg
```

Note that to terminate options processing still requires a double dash `--`.

`GetOptions()` will not interpret a leading `< "<"` as option starters if the next argument is a reference. To force `< "<"` and `< ""` as option starters, use `< "<"`. Confusing? Well, **using a starter argument is strongly deprecated** anyway.

Configuration variables

Previous versions of `Getopt::Long` used variables for the purpose of configuring. Although manipulating these variables still work, it is strongly encouraged to use the `Configure` routine that was introduced in version 2.17. Besides, it is much easier.

Trouble Shooting

Warning: Ignoring '!' modifier for short option

This warning is issued when the `!` modifier is applied to a short (one-character) option and bundling is in effect. E.g.,

```
Getopt::Long::Configure("bundling");
GetOptions("foo|f!" => \$foo);
```

Note that older `Getopt::Long` versions did not issue a warning, because the `!` modifier was applied to the first name only. This bug was fixed in 2.22.

Solution: separate the long and short names and apply the `!` to the long names only, e.g.,

```
GetOptions("foo!" => \$foo, "f" => \$foo);
```

GetOptions does not return a false result when an option is not supplied

That's why they're called 'options'.

GetOptions does not split the command line correctly

The command line is not split by `GetOptions`, but by the command line interpreter (CLI). On Unix, this is the shell. On Windows, it is `COMMAND.COM` or `CMD.EXE`. Other operating systems have other CLIs.

It is important to know that these CLIs may behave different when the command line contains special characters, in particular quotes or backslashes. For example, with Unix shells you can use single quotes (`'`) and double quotes (`"`) to group words together. The following alternatives are equivalent on Unix:

```
"two words"
'two words'
two\ words
```

In case of doubt, insert the following statement in front of your Perl program:

```
print STDERR (join(" | ", @ARGV), "\n");
```

to verify how your CLI passes the arguments to the program.

How do I put a "-?" option into a Getopt::Long?

You can only obtain this using an alias, and Getopt::Long of at least version 2.13.

```
use Getopt::Long;  
GetOptions ("help|?");    # -help and -? will both set $opt_help
```

AUTHOR

Johan Vromans <jvromans@squirrel.nl>

COPYRIGHT AND DISCLAIMER

This program is Copyright 2002,1990 by Johan Vromans. This program is free software; you can redistribute it and/or modify it under the terms of the Perl Artistic License or the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

If you do not have a copy of the GNU General Public License write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

NAME

`getopt` – Process single-character switches with switch clustering

`getopts` – Process single-character switches with switch clustering

SYNOPSIS

```
use Getopt::Std;

getopt('oDI');    # -o, -D & -I take arg.  Sets $opt_* as a side effect.
getopt('oDI', \%opts);    # -o, -D & -I take arg.  Values in %opts
getopts('oif:');  # -o & -i are boolean flags, -f takes an argument
                  # Sets $opt_* as a side effect.
getopts('oif:', \%opts); # options as above. Values in %opts
```

DESCRIPTION

The `getopt()` function processes single-character switches with switch clustering. Pass one argument which is a string containing all switches that take an argument. For each switch found, sets `$opt_x` (where `x` is the switch name) to the value of the argument if an argument is expected, or 1 otherwise. Switches which take an argument don't care whether there is a space between the switch and the argument.

The `getopts()` function is similar, but you should pass to it the list of all switches to be recognized. If unspecified switches are found on the command-line, the user will be warned that an unknown option was given.

Note that, if your code is running under the recommended `use strict 'vars'` pragma, you will need to declare these package variables with "our":

```
our($opt_x, $opt_y);
```

For those of you who don't like additional global variables being created, `getopt()` and `getopts()` will also accept a hash reference as an optional second argument. Hash keys will be `x` (where `x` is the switch name) with key values the value of the argument or 1 if no argument is specified.

To allow programs to process arguments that look like switches, but aren't, both functions will stop processing switches when they see the argument `—`. The `—` will be removed from `@ARGV`.

NAME

Hash::Util – A selection of general-utility hash subroutines

SYNOPSIS

```
use Hash::Util qw(lock_keys  unlock_keys
                  lock_value  unlock_value
                  lock_hash   unlock_hash);

%hash = (foo => 42, bar => 23);
lock_keys(%hash);
lock_keys(%hash, @keyset);
unlock_keys(%hash);

lock_value(%hash, 'foo');
unlock_value(%hash, 'foo');

lock_hash(%hash);
unlock_hash(%hash);
```

DESCRIPTION

Hash::Util contains special functions for manipulating hashes that don't really warrant a keyword.

By default Hash::Util does not export anything.

Restricted hashes

5.8.0 introduces the ability to restrict a hash to a certain set of keys. No keys outside of this set can be added.

It also introduces the ability to lock an individual key so it cannot be deleted and the value cannot be changed.

This is intended to largely replace the deprecated pseudo-hashes.

lock_keys**unlock_keys**

```
lock_keys(%hash);
lock_keys(%hash, @keys);
```

Restricts the given %hash's set of keys to @keys. If @keys is not given it restricts it to its current keyset. No more keys can be added. `delete()` and `exists()` will still work, but it does not effect the set of allowed keys.

```
unlock_keys(%hash);
```

Removes the restriction on the %hash's keyset.

lock_value**unlock_value**

```
lock_key(%hash, $key);
unlock_key(%hash, $key);
```

Locks and unlocks an individual key of a hash. The value of a locked key cannot be changed.

%hash must have already been locked for this to have useful effect.

lock_hash**unlock_hash**

```
lock_hash(%hash);
```

`lock_hash()` locks an entire hash, making all keys and values readonly. No value can be changed, no keys can be added or deleted.

```
unlock_hash(%hash);
```


`unlock_hash()` does the opposite of `lock_hash()`. All keys and values are made read/write. All values can be changed and keys can be added and deleted.

AUTHOR

Michael G Schwern <schwern@pobox.com> on top of code by Nick Ing-Simmons and Jeffrey Friedl.

SEE ALSO

[Scalar::Util](#), [List::Util](#), [Hash::Util](#)

NAME

I18N::Collate – compare 8-bit scalar data according to the current locale

SYNOPSIS

```
use I18N::Collate;
setlocale(LC_COLLATE, 'locale-of-your-choice');
$s1 = new I18N::Collate "scalar_data_1";
$s2 = new I18N::Collate "scalar_data_2";
```

DESCRIPTION

WARNING: starting from the Perl version 5.003_06
the I18N::Collate interface for comparing 8-bit scalar data
according to the current locale

HAS BEEN DEPRECATED

That is, please do not use it anymore for any new applications
and please migrate the old applications away from it because its
functionality was integrated into the Perl core language in the
release 5.003_06.

See the perllocale manual page for further information.

This module provides you with objects that will collate according to your national character set, provided
that the POSIX `setlocale()` function is supported on your system.

You can compare `$s1` and `$s2` above with

```
$s1 le $s2
```

to extract the data itself, you'll need a dereference: `$$s1`

This module uses `POSIX::setlocale()`. The basic collation conversion is done by `strxfrm()` which
terminates at NUL characters being a decent C routine. `collate_xfrm()` handles embedded NUL
characters gracefully.

The available locales depend on your operating system; try whether `locale -a` shows them or man pages
for "locale" or "nlsinfo" or the direct approach `ls /usr/lib/nls/loc` or `ls /usr/lib/nls` or `ls /usr/lib/locale`. Not all the locales that your vendor supports are necessarily installed: please consult
your operating system's documentation and possibly your local system administration. The locale names are
probably something like `xx_XX.(ISO)?8859-N` or `xx_XX.(ISO)?8859N`, for example
`fr_CH.ISO8859-1` is the Swiss (CH) variant of French (fr), ISO Latin (8859) 1 (-1) which is the Western
European character set.

NAME

I18N::Langinfo – query locale information

SYNOPSIS

```
use I18N::Langinfo;
```

DESCRIPTION

The `langinfo()` function queries various locale information that can be used to localize output and user interfaces. The `langinfo()` requires one numeric argument that identifies the locale constant to query: if no argument is supplied, `$_` is used. The numeric constants appropriate to be used as arguments are exportable from `I18N::Langinfo`.

The following example will import the `langinfo()` function itself and three constants to be used as arguments to `langinfo()`: a constant for the abbreviated first day of the week (the numbering starts from Sunday = 1) and two more constants for the affirmative and negative answers for a yes/no question in the current locale.

```
use I18N::Langinfo qw(langinfo ABDAY_1 YESSTR NOSTR);
my ($abday_1, $yesstr, $nostr) = map { langinfo } qw(ABDAY_1 YESSTR NOSTR);
print "$abday_1? [$yesstr/$nostr] ";
```

In other words, in the "C" (or English) locale the above will probably print something like:

```
Sun? [yes/no]
```

but under a French locale

```
dim? [oui/non]
```

The usually available constants are

```
ABDAY_1 ABDAY_2 ABDAY_3 ABDAY_4 ABDAY_5 ABDAY_6 ABDAY_7
ABMON_1 ABMON_2 ABMON_3 ABMON_4 ABMON_5 ABMON_6
ABMON_7 ABMON_8 ABMON_9 ABMON_10 ABMON_11 ABMON_12
DAY_1 DAY_2 DAY_3 DAY_4 DAY_5 DAY_6 DAY_7
MON_1 MON_2 MON_3 MON_4 MON_5 MON_6
MON_7 MON_8 MON_9 MON_10 MON_11 MON_12
```

for abbreviated and full length days of the week and months of the year,

```
D_T_FMT D_FMT T_FMT
```

for the date–time, date, and time formats used by the `strftime()` function (see [POSIX](#))

```
AM_STR PM_STR T_FMT_AMPM
```

for the locales for which it makes sense to have ante meridiem and post meridiem time formats,

```
CODESET CRNCYSTR RADIXCHAR
```

for the character code set being used (such as "ISO8859-1", "cp850", "koi8-r", "sjis", "utf8", etc.), for the currency string, for the radix character used between the integer and the fractional part of decimal numbers (yes, this is redundant with `POSIX::localeconv()`)

```
YESSTR YESEXPR NOSTR NOEXPR
```

for the affirmative and negative responses and expressions, and

```
ERA ERA_D_FMT ERA_D_T_FMT ERA_T_FMT
```

for the Japanese Emperor eras (naturally only defined under Japanese locales).

See your [langinfo\(3\)](#) for more information about the available constants. (Often this means having to look

directly at the *langinfo.h* C header file.)

Note that unfortunately none of the above constants are guaranteed to be available on a particular platform. To be on the safe side you can wrap the import in an eval like this:

```
eval {  
    require I18N::Langinfo;  
    I18N::Langinfo->import(qw(langinfo CODESET));  
    $codeset = langinfo(CODESET()); # note the ()  
};  
if (!$?) { ... failed ... }
```

EXPORT

Nothing is exported by default.

SEE ALSO

[perllocale](#), [localeconv](#), [setlocale](#), [nl_langinfo\(3\)](#).

The `langinfo()` is just a wrapper for the C `nl_langinfo()` interface.

AUTHOR

Jarkko Hietaniemi, <jhi@hut.fi>

COPYRIGHT AND LICENSE

Copyright 2001 by Jarkko Hietaniemi

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

I18N::LangTags – functions for dealing with RFC3066–style language tags

SYNOPSIS

```
use I18N::LangTags qw(is_language_tag same_language_tag
                      extract_language_tags super_languages
                      similarity_language_tag is_dialect_of
                      locale2language_tag alternate_language_tags
                      encode_language_tag panic_languages
                      );
```

...or whatever of those functions you want to import. Those are all the exportable functions — you're free to import only some, or none at all. By default, none are imported. If you say:

```
use I18N::LangTags qw(:ALL)
```

...then all are exported. (This saves you from having to use something less obvious like `use I18N::LangTags qw(/./)`.)

If you don't import any of these functions, assume a `&I18N::LangTags::` in front of all the function names in the following examples.

DESCRIPTION

Language tags are a formalism, described in RFC 3066 (obsoleting 1766), for declaring what language form (language and possibly dialect) a given chunk of information is in.

This library provides functions for common tasks involving language tags as they are needed in a variety of protocols and applications.

Please see the "See Also" references for a thorough explanation of how to correctly use language tags.

- the function `is_language_tag($lang1)`

Returns true iff `$lang1` is a formally valid language tag.

```
is_language_tag("fr")           is TRUE
is_language_tag("x-jicarilla")  is FALSE
    (Subtags can be 8 chars long at most -- 'jicarilla' is 9)

is_language_tag("sgn-US")       is TRUE
    (That's American Sign Language)

is_language_tag("i-Klikitat")   is TRUE
    (True without regard to the fact noone has actually
    registered Klikitat -- it's a formally valid tag)

is_language_tag("fr-patois")    is TRUE
    (Formally valid -- altho descriptively weak!)

is_language_tag("Spanish")      is FALSE
is_language_tag("french-patois") is FALSE
    (No good -- first subtag has to match
    /^[xXiI]|[a-zA-Z]{2,3}$/ -- see RFC3066)

is_language_tag("x-borg-prot2532") is TRUE
    (Yes, subtags can contain digits, as of RFC3066)
```

- the function `extract_language_tags($whatever)`

Returns a list of whatever looks like formally valid language tags in `$whatever`. Not very smart, so don't get too creative with what you want to feed it.

```
extract_language_tags("fr, fr-ca, i-mingo")
returns: ('fr', 'fr-ca', 'i-mingo')
```

```
extract_language_tags("It's like this: I'm in fr -- French!")
returns:    ('It', 'in', 'fr')
(So don't just feed it any old thing.)
```

The output is untainted. If you don't know what tainting is, don't worry about it.

- the function `same_language_tag($lang1, $lang2)`

Returns true iff `$lang1` and `$lang2` are acceptable variant tags representing the same language-form.

```
same_language_tag('x-kadara', 'i-kadara')  is TRUE
(The x/i- alternation doesn't matter)
same_language_tag('X-KADARA', 'i-kadara')  is TRUE
(...and neither does case)
same_language_tag('en', 'en-US')           is FALSE
(all-English is not the SAME as US English)
same_language_tag('x-kadara', 'x-kadar')   is FALSE
(these are totally unrelated tags)
same_language_tag('no-bok', 'nb')          is TRUE
(no-bok is a legacy tag for nb (Norwegian Bokmal))
```

`same_language_tag` works by just seeing whether `encode_language_tag($lang1)` is the same as `encode_language_tag($lang2)`.

(Yes, I know this function is named a bit oddly. Call it historic reasons.)

- the function `similarity_language_tag($lang1, $lang2)`

Returns an integer representing the degree of similarity between tags `$lang1` and `$lang2` (the order of which does not matter), where similarity is the number of common elements on the left, without regard to case and to x/i- alternation.

```
similarity_language_tag('fr', 'fr-ca')      is 1
(one element in common)
similarity_language_tag('fr-ca', 'fr-FR')   is 1
(one element in common)
similarity_language_tag('fr-CA-joual',
                        'fr-CA-PEI')        is 2
similarity_language_tag('fr-CA-joual', 'fr-CA') is 2
(two elements in common)
similarity_language_tag('x-kadara', 'i-kadara') is 1
(x/i- doesn't matter)
similarity_language_tag('en', 'x-kadar')    is 0
similarity_language_tag('x-kadara', 'x-kadar') is 0
(unrelated tags -- no similarity)
similarity_language_tag('i-cree-syllabic',
                        'i-cherokee-syllabic') is 0
(no B<leftmost> elements in common!)
```

- the function `is_dialect_of($lang1, $lang2)`

Returns true iff language tag `$lang1` represents a subform of language tag `$lang2`.

Get the order right! It doesn't work the other way around!

```
is_dialect_of('en-US', 'en')                is TRUE
(American English IS a dialect of all-English)
is_dialect_of('fr-CA-joual', 'fr-CA')       is TRUE
```

```

is_dialect_of('fr-CA-joual', 'fr')      is TRUE
    (Joual is a dialect of (a dialect of) French)

is_dialect_of('en', 'en-US')            is FALSE
    (all-English is a NOT dialect of American English)

is_dialect_of('fr', 'en-CA')            is FALSE

is_dialect_of('en', 'en')                is TRUE
is_dialect_of('en-US', 'en-US')         is TRUE
    (B<Note:> these are degenerate cases)

is_dialect_of('i-mingo-tom', 'x-Mingo') is TRUE
    (the x/i thing doesn't matter, nor does case)

is_dialect_of('nn', 'no')                is TRUE
    (because 'nn' (New Norse) is aliased to 'no-nyn',
     as a special legacy case, and 'no-nyn' is a
     subform of 'no' (Norwegian))

```

- the function `super_languages($lang1)`

Returns a list of language tags that are superordinate tags to `$lang1` — it gets this by removing subtags from the end of `$lang1` until nothing (or just "i" or "x") is left.

```

super_languages("fr-CA-joual") is ("fr-CA", "fr")
super_languages("en-AU")      is ("en")
super_languages("en")         is empty-list, ()
super_languages("i-cherokee") is empty-list, ()
...not ("i"), which would be illegal as well as pointless.

```

If `$lang1` is not a valid language tag, returns empty-list in a list context, undef in a scalar context.

A notable and rather unavoidable problem with this method: "x-mingo-tom" has an "x" because the whole tag isn't an IANA-registered tag — but `super_languages('x-mingo-tom')` is ('x-mingo') — which isn't really right, since 'i-mingo' is registered. But this module has no way of knowing that. (But note that `same_language_tag('x-mingo', 'i-mingo')` is TRUE.)

More importantly, you assume *at your peril* that superordinates of `$lang1` are mutually intelligible with `$lang1`. Consider this carefully.

- the function `locale2language_tag($locale_identifier)`

This takes a locale name (like "en", "en-US", or "en-US.ISO8859-1") and maps it to a language tag. If it's not mappable (as with, notably, "C" and "POSIX"), this returns empty-list in a list context, or undef in a scalar context.

```

locale2language_tag("en") is "en"
locale2language_tag("en-US") is "en-US"
locale2language_tag("en-US.ISO8859-1") is "en-US"
locale2language_tag("C") is undef or ()
locale2language_tag("POSIX") is undef or ()
locale2language_tag("POSIX") is undef or ()

```

I'm not totally sure that locale names map satisfactorily to language tags. Think REAL hard about how you use this. YOU HAVE BEEN WARNED.

The output is untainted. If you don't know what tainting is, don't worry about it.

- the function `encode_language_tag($lang1)`

This function, if given a language tag, returns an encoding of it such that:

- * tags representing different languages never get the same encoding.
- * tags representing the same language always get the same encoding.
- * an encoding of a formally valid language tag always is a string value that is defined, has length, and is true if considered as a boolean.

Note that the encoding itself is **not** a formally valid language tag. Note also that you cannot, currently, go from an encoding back to a language tag that it's an encoding of.

Note also that you **must** consider the encoded value as atomic; i.e., you should not consider it as anything but an opaque, unanalysable string value. (The internals of the encoding method may change in future versions, as the language tagging standard changes over time.)

`encode_language_tag` returns undef if given anything other than a formally valid language tag.

The reason `encode_language_tag` exists is because different language tags may represent the same language; this is normally treatable with `same_language_tag`, but consider this situation:

You have a data file that expresses greetings in different languages. Its format is "[language tag]=[how to say 'Hello']", like:

```
en-US=Hiho
fr=Bonjour
i-mingo=Hau'
```

And suppose you write a program that reads that file and then runs as a daemon, answering client requests that specify a language tag and then expect the string that says how to greet in that language. So an interaction looks like:

```
greeting-client asks:   fr
greeting-server answers: Bonjour
```

So far so good. But suppose the way you're implementing this is:

```
my %greetings;
die unless open(IN, "<in.dat");
while(<IN>) {
    chomp;
    next unless /^([^=]+)=(.+)$/s;
    my($lang, $expr) = ($1, $2);
    $greetings{$lang} = $expr;
}
close(IN);
```

at which point `%greetings` has the contents:

```
"en-US"    => "Hiho"
"fr"        => "Bonjour"
"i-mingo"   => "Hau' "
```

And suppose then that you answer client requests for language `$wanted` by just looking up `$greetings{$wanted}`.

If the client asks for "fr", that will look up successfully in `%greetings`, to the value "Bonjour". And if the client asks for "i-mingo", that will look up successfully in `%greetings`, to the value "Hau'".

But if the client asks for "i-Mingo" or "x-mingo", or "Fr", then the lookup in `%greetings` fails. That's the Wrong Thing.

You could instead do lookups on `$wanted` with:

```
use I18N::LangTags qw(same_language_tag);
my $response = '';
foreach my $l2 (keys %greetings) {
    if(same_language_tag($wanted, $l2)) {
        $response = $greetings{$l2};
        last;
    }
}
```

But that's rather inefficient. A better way to do it is to start your program with:

```
use I18N::LangTags qw(encode_language_tag);
my %greetings;
die unless open(IN, "<in.dat");
while(<IN>) {
    chomp;
    next unless /^(^[^=]+)=(.+)$/s;
    my($lang, $expr) = ($1, $2);
    $greetings{
        encode_language_tag($lang)
    } = $expr;
}
close(IN);
```

and then just answer client requests for language `$wanted` by just looking up

```
$greetings{encode_language_tag($wanted)}
```

And that does the Right Thing.

- the function `alternate_language_tags($lang1)`

This function, if given a language tag, returns all language tags that are alternate forms of this language tag. (I.e., tags which refer to the same language.) This is meant to handle legacy tags caused by the minor changes in language tag standards over the years; and the `x-/i-` alternation is also dealt with.

Note that this function does *not* try to equate new (and never-used, and unusable) ISO639-2 three-letter tags to old (and still in use) ISO639-1 two-letter equivalents — like "ara" — "ar" — because "ara" has *never* been in use as an Internet language tag, and RFC 3066 stipulates that it never should be, since a shorter tag ("ar") exists.

Examples:

```
alternate_language_tags('no-bok')    is ('nb')
alternate_language_tags('nb')        is ('no-bok')
alternate_language_tags('he')        is ('iw')
alternate_language_tags('iw')        is ('he')
alternate_language_tags('i-hakka')    is ('zh-hakka', 'x-hakka')
alternate_language_tags('zh-hakka')   is ('i-hakka', 'x-hakka')
alternate_language_tags('en')         is ()
alternate_language_tags('x-mingo-tom') is ('i-mingo-tom')
alternate_language_tags('x-klikitat') is ('i-klikitat')
alternate_language_tags('i-klikitat') is ('x-klikitat')
```

This function returns empty-list if given anything other than a formally valid language tag.

- the function `@langs = panic_languages(@accept_languages)`

This function takes a list of 0 or more language tags that constitute a given user's Accept-Language list, and returns a list of tags for *other* (non-super) languages that are probably acceptable to the user,

to be used *if all else fails*.

For example, if a user accepts only 'ca' (Catalan) and 'es' (Spanish), and the documents/interfaces you have available are just in German, Italian, and Chinese, then the user will most likely want the Italian one (and not the Chinese or German one!), instead of getting nothing. So `panic_languages('ca', 'es')` returns a list containing 'it' (Italian).

English ('en') is *always* in the return list, but whether it's at the very end or not depends on the input languages. This function works by consulting an internal table that stipulates what common languages are "close" to each other.

A useful construct you might consider using is:

```
@fallbacks = super_languages(@accept_languages);
push @fallbacks, panic_languages(
    @accept_languages, @fallbacks,
);
```

ABOUT LOWERCASING

I've considered making all the above functions that output language tags return all those tags strictly in lowercase. Having all your language tags in lowercase does make some things easier. But you might as well just lowercase as you like, or call `encode_language_tag($lang1)` where appropriate.

ABOUT UNICODE PLAINTEXT LANGUAGE TAGS

In some future version of I18N::LangTags, I plan to include support for RFC2482-style language tags — which are basically just normal language tags with their ASCII characters shifted into Plane 14.

SEE ALSO

- * [I18N::LangTags::List/I18N::LangTags::List](#)
- * RFC 3066, <ftp://ftp.isi.edu/in-notes/rfc3066.txt>, "Tags for the Identification of Languages". (Obsoletes RFC 1766)
- * RFC 2277, <ftp://ftp.isi.edu/in-notes/rfc2277.txt>, "IETF Policy on Character Sets and Languages".
- * RFC 2231, <ftp://ftp.isi.edu/in-notes/rfc2231.txt>, "MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations".
- * RFC 2482, <ftp://ftp.isi.edu/in-notes/rfc2482.txt>, "Language Tagging in Unicode Plain Text".
- * Locale::Codes, in <http://www.perl.com/CPAN/modules/by-module/Locale/>
- * ISO 639, "Code for the representation of names of languages", <http://www.indigo.ie/egt/standards/iso639/iso639-1-en.html>
- * ISO 639-2, "Codes for the representation of names of languages", including three-letter codes, <http://lcweb.loc.gov/standards/iso639-2/bibcodes.html>
- * The IANA list of registered languages (hopefully up-to-date), <ftp://ftp.isi.edu/in-notes/iana/assignments/languages/>

COPYRIGHT

Copyright (c) 1998–2001 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

The programs and documentation in this dist are distributed in the hope that they will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Sean M. Burke sburke@cpan.org

NAME

I18N::LangTags::List — tags and names for human languages

SYNOPSIS

```
use I18N::LangTags::List;
print "Parlez-vous... ", join(' ',
    I18N::LangTags::List::name('elx') || 'unknown_language',
    I18N::LangTags::List::name('ar-Kw') || 'unknown_language',
    I18N::LangTags::List::name('en') || 'unknown_language',
    I18N::LangTags::List::name('en-CA') || 'unknown_language',
), "?\n";
```

prints:

```
Parlez-vous... Elamite, Kuwait Arabic, English, Canadian English?
```

DESCRIPTION

This module provides a function `I18N::LangTags::List::name(langtag)` that takes a language tag (see [I18N::LangTags/I18N::LangTags](#)) and returns the best attempt at an English name for it, or undef if it can't make sense of the tag.

The function `I18N::LangTags::List::name(...)` is not exported.

The map of tags-to-names that it uses is accessible as `%I18N::LangTags::List::Name`, and it's the same as the list that follows in this documentation, which should be useful to you even if you don't use this module.

ABOUT LANGUAGE TAGS

Internet language tags, as defined in RFC 3066, are a formalism for denoting human languages. The two-letter ISO 639-1 language codes are well known (as "en" for English), as are their forms when qualified by a country code ("en-US"). Less well-known are the arbitrary-length non-ISO codes (like "i-mingo"), and the recently (in 2001) introduced three-letter ISO-639-2 codes.

Remember these important facts:

- Language tags are not locale IDs. A locale ID is written with a "_" instead of a "-", (almost?) always matches `m/^\w\w_\w\w\b/`, and *means* something different than a language tag. A language tag denotes a language. A locale ID denotes a language *as used in* a particular place, in combination with non-linguistic location-specific information such as what currency is used there. Locales *also* often denote character set information, as in "en_US.ISO8859-1".
- Language tags are not for computer languages.
- "Dialect" is not a useful term, since there is no objective criterion for establishing when two language-forms are dialects of eachother, or are separate languages.
- Language tags are not case-sensitive. en-US, en-us, En-U_s, etc., are all the same tag, and denote the same language.
- Not every language tag really refers to a single language. Some language tags refer to conditions: i-default (system-message text in English plus maybe other languages), und (undetermined language). Others (notably lots of the three-letter codes) are bibliographic tags that classify whole groups of languages, as with cus "Cushitic (Other)" (i.e., a language that has been classed as Cushtic, but which has no more specific code) or the even less linguistically coherent sai for "South American Indian (Other)". Though useful in bibliography, **SUCH TAGS ARE NOT FOR GENERAL USE**. For further guidance, email me.
- Language tags are not country codes. In fact, they are often distinct codes, as with language tag ja for Japanese, and ISO 3166 country code .jp for Japan.

LIST OF LANGUAGES

The first part of each item is the language tag, between {...}. It is followed by an English name for the language or language-group. Language tags that I judge to be not for general use, are bracketed.

This list is in alphabetical order by English name of the language.

=for reminder

The name in the =item line MUST NOT have 's in it!!

=for woohah START

{ab} : Abkhazian

eq Abkhaz

{ace} : Achinese

{ach} : Acoli

{ada} : Adangme

{aa} : Afar

{afh} : Afrihili

(Artificial)

{af} : Afrikaans

[{afa} : Afro-Asiatic (Other)]

{aka} : Akan

{akk} : Akkadian

(Historical)

{sq} : Albanian

{ale} : Aleut

[{alg} : Algonquian languages]

NOT Algonquin!

[{tut} : Altaic (Other)]

{am} : Amharic

NOT Aramaic!

{i-ami} : Ami

eq Amis. eq 'Amis. eq Pangca.

[{apa} : Apache languages]

{ar} : Arabic

Many forms are mutually un-intelligible in spoken media. Notable forms: {ar-ae} UAE Arabic; {ar-bh} Bahrain Arabic; {ar-dz} Algerian Arabic; {ar-eg} Egyptian Arabic; {ar-iq} Iraqi Arabic; {ar-jo} Jordanian Arabic; {ar-kw} Kuwait Arabic; {ar-lb} Lebanese Arabic; {ar-ly} Libyan Arabic; {ar-ma} Moroccan Arabic; {ar-om} Omani Arabic; {ar-qa} Qatari Arabic; {ar-sa} Sauda Arabic; {ar-sy} Syrian Arabic; {ar-tn} Tunisian Arabic; {ar-ye} Yemen Arabic.

{arc} : Aramaic

NOT Amharic! NOT Samaritan Aramaic!

{arp} : Arapaho

{arn} : Araucanian

{arw} : Arawak

{hy} : Armenian

[{art} : Artificial (Other)]

{as} : Assamese

[{ath} : Athapaskan languages]

eq Athabaskan. eq Athapaskan. eq Athabaskan.

[{aus} : Australian languages]
[{map} : Austronesian (Other)]
{ava} : Avaric
{ae} : Avestan
 eq Zend

{awa} : Awadhi
{ay} : Aymara
{az} : Azerbaijani
 eq Azeri

{ban} : Balinese
[{bat} : Baltic (Other)]
{bal} : Baluchi
{bam} : Bambara
[{bai} : Bamileke languages]
{bad} : Banda
[{bnt} : Bantu (Other)]
{bas} : Basa
{ba} : Bashkir
{eu} : Basque
{btk} : Batak (Indonesia)
{bej} : Beja
{be} : Belarusian
 eq Belarussian. eq Byelarussian. eq Belorussian. eq Byelorussian. eq White Russian. eq White
 Ruthenian. NOT Ruthenian!

{bem} : Bemba
{bn} : Bengali
 eq Bangla.

[{ber} : Berber (Other)]
{bho} : Bhojpuri
{bh} : Bihari
{bik} : Bikol
{bin} : Bini
{bi} : Bislama
 eq Bichelamar.

{bs} : Bosnian
{bra} : Braj
{br} : Breton
{bug} : Buginese
{bg} : Bulgarian
{i-bnn} : Bunun
{bua} : Buriat
{my} : Burmese
{cad} : Caddo
{car} : Carib
{ca} : Catalan
 eq Catalán. eq Catalanian.

[{cau} : Caucasian (Other)]
{ceb} : Cebuano

[{cel} : Celtic (Other)]

Notable forms: {cel-gaulish} Gaulish (Historical)

[{cai} : Central American Indian (Other)]

{chg} : Chagatai

(Historical?)

[{cmc} : Chamic languages]

{ch} : Chamorro

{ce} : Chechen

{chr} : Cherokee

eq Tsalagi

{chy} : Cheyenne

{chb} : Chibcha

(Historical) NOT Chibchan (which is a language family).

{ny} : Chichewa

eq Nyanja. eq Chinyanja.

{zh} : Chinese

Many forms are mutually un-intelligible in spoken media. Notable subforms: {zh-cn} PRC Chinese; {zh-hk} Hong Kong Chinese; {zh-mo} Macau Chinese; {zh-sg} Singapore Chinese; {zh-tw} Taiwan Chinese; {zh-guoyu} Mandarin [Putonghua/Guoyu]; {zh-hakka} Hakka [formerly i-hakka]; {zh-min} Hokkien; {zh-min-nan} Southern Hokkien; {zh-wuu} Shanghaiese; {zh-xiang} Hunanese; {zh-gan} Gan; {zh-yue} Cantonese.

=for etc {i-hakka} Hakka (old tag)

{chn} : Chinook Jargon

eq Chinook Wawa.

{chp} : Chipewyan

{cho} : Choctaw

{cu} : Church Slavic

eq Old Church Slavonic.

{chk} : Chuukese

eq Trukese. eq Chuuk. eq Truk. eq Ruk.

{cv} : Chuvash

{cop} : Coptic

{kw} : Cornish

{co} : Corsican

eq Corse.

{cre} : Cree

NOT Creek!

{mus} : Creek

NOT Cree!

[{cpe} : English-based Creoles and pidgins (Other)]

[{cpf} : French-based Creoles and pidgins (Other)]

[{cpp} : Portuguese-based Creoles and pidgins (Other)]

[{crp} : Creoles and pidgins (Other)]

{hr} : Croatian

eq Croat.

[{cus} : Cushitic (Other)]

{cs} : Czech

{dak} : Dakota

eq Nakota. eq Latoka.

{da} : Danish

{day} : Dayak

{i-default} : Default (Fallthru) Language

Defined in RFC 2277, this is for tagging text (which must include English text, and might/should include text in other appropriate languages) that is emitted in a context where language-negotiation wasn't possible — in SMTP mail failure messages, for example.

{del} : Delaware

{din} : Dinka

{div} : Divehi

{doi} : Dogri

NOT Dogrib!

{dgr} : Dogrib

NOT Dogri!

[{dra} : Dravidian (Other)]

{dua} : Duala

{nl} : Dutch

eq Netherlander. Notable forms: {nl-nl} Netherlands Dutch; {nl-be} Belgian Dutch.

{dum} : Middle Dutch (ca.1050–1350)

(Historical)

{dyu} : Dyula

{dz} : Dzongkha

{efi} : Efik

{egy} : Ancient Egyptian

(Historical)

{eka} : Ekajuk

{elx} : Elamite

(Historical)

{en} : English

Notable forms: {en-au} Australian English; {en-bz} Belize English; {en-ca} Canadian English; {en-gb} UK English; {en-ie} Irish English; {en-jm} Jamaican English; {en-nz} New Zealand English; {en-ph} Philippine English; {en-tt} Trinidad English; {en-us} US English; {en-za} South African English; {en-zw} Zimbabwe English.

{enm} : Old English (1100–1500)

(Historical)

{ang} : Old English (ca.450–1100)

eq Anglo-Saxon. (Historical)

{eo} : Esperanto

(Artificial)

{et} : Estonian

{ewe} : Ewe

{ewo} : Ewondo
{fan} : Fang
{fat} : Fanti
{fo} : Faroese
{fj} : Fijian
{fi} : Finnish
[{fiu} : Finno-Ugrian (Other)]
 eq Finno-Ugric. NOT Ugaritic!

{fon} : Fon
{fr} : French
 Notable forms: {fr-fr} France French; {fr-be} Belgian French; {fr-ca} Canadian French; {fr-ch} Swiss French; {fr-lu} Luxembourg French; {fr-mc} Monaco French.

{frm} : Middle French (ca.1400–1600)
 (Historical)

{fro} : Old French (842–ca.1400)
 (Historical)

{fy} : Frisian
{fur} : Friulian
{ful} : Fulah
{gaa} : Ga
{gd} : Scots Gaelic
 NOT Scots!

{gl} : Gallegan
 eq Galician

{lug} : Ganda
{gay} : Gayo
{gba} : Gbaya
{gez} : Geez
 eq Ge'ez

{ka} : Georgian
{de} : German
 Notable forms: {de-at} Austrian German; {de-be} Belgian German; {de-ch} Swiss German; {de-de} Germany German; {de-li} Liechtenstein German; {de-lu} Luxembourg German.

{gmh} : Middle High German (ca.1050–1500)
 (Historical)

{goh} : Old High German (ca.750–1050)
 (Historical)

[{gem} : Germanic (Other)]
{gil} : Gilbertese
{gon} : Gondi
{gor} : Gorontalo
{got} : Gothic
 (Historical)

{grb} : Grebo
{grc} : Ancient Greek
 (Historical) (Until 15th century or so.)

{el} : Modern Greek
(Since 15th century or so.)

{gn} : Guarani
Guaraní

{gu} : Gujarati

{gwi} : Gwich'in
eq Gwichin

{hai} : Haida

{ha} : Hausa

{haw} : Hawaiian
Hawai'ian

{he} : Hebrew
(Formerly "iw".)
=for etc {iw} Hebrew (old tag)

{hz} : Herero

{hil} : Hiligaynon

{him} : Himachali

{hi} : Hindi

{ho} : Hiri Motu

{hit} : Hittite
(Historical)

{hmn} : Hmong

{hu} : Hungarian

{hup} : Hupa

{iba} : Iban

{is} : Icelandic

{ibo} : Igbo

{ijo} : Ijo

{ilo} : Iloko

[{inc} : Indic (Other)]

[{ine} : Indo-European (Other)]

{id} : Indonesian
(Formerly "in".)
=for etc {in} Indonesian (old tag)

{ia} : Interlingua (International Auxiliary Language Association)
(Artificial) NOT Interlingue!

{ie} : Interlingue
(Artificial) NOT Interlingua!

{iu} : Inuktitut
A subform of "Eskimo".

{ik} : Inupiaq
A subform of "Eskimo".

[{ira} : Iranian (Other)]

{ga} : Irish

{mga} : Middle Irish (900–1200)

(Historical)

{sga} : Old Irish (to 900)

(Historical)

[{iro} : Iroquoian languages]

{it} : Italian

Notable forms: {it-it} Italy Italian; {it-ch} Swiss Italian.

{ja} : Japanese

(NOT "jp"!)

{jw} : Javanese

{jrb} : Judeo–Arabic

{jpr} : Judeo–Persian

{kab} : Kabyle

{kac} : Kachin

{kl} : Kalaallisut

eq Greenlandic "Eskimo"

{kam} : Kamba

{kn} : Kannada

eq Kanarese. NOT Canadian!

{kau} : Kanuri

{kaa} : Kara–Kalpak

{kar} : Karen

{ks} : Kashmiri

{kaw} : Kawi

{kk} : Kazakh

{kha} : Khasi

{km} : Khmer

eq Cambodian. eq Kampuchean.

[{khi} : Khoisan (Other)]

{kho} : Khotanese

{ki} : Kikuyu

eq Gikuyu.

{kmb} : Kimbundu

{rw} : Kinyarwanda

{ky} : Kirghiz

{i-klíngon} : Klingon

{kv} : Komi

{kon} : Kongo

{kok} : Konkani

{ko} : Korean

{kos} : Kosraean

{kpe} : Kpelle

{kro} : Kru

{kj} : Kuanyama

{kum} : Kumyk

{ku} : Kurdish

{kru} : Kurukh

{kut} : Kutenai
{lad} : Ladino
eq Judeo–Spanish. NOT Ladin (a minority language in Italy).
{lah} : Lahnda
NOT Lamba!
{lam} : Lamba
NOT Lahnda!
{lo} : Lao
eq Laotian.
{la} : Latin
(Historical) NOT Ladin! NOT Ladino!
{lv} : Latvian
eq Lettish.
{lb} : Letzeburgesch
eq Luxemburgian, eq Luxemburger. (Formerly i–lux.)
=for etc {i–lux} Letzeburgesch (old tag)
{lez} : Lezghian
{ln} : Lingala
{lt} : Lithuanian
{nds} : Low German
eq Low Saxon. eq Low German. eq Low Saxon.
{loz} : Lozi
{lub} : Luba–Katanga
{lua} : Luba–Lulua
{lui} : Luiseno
eq Luiseño.
{lun} : Lunda
{luo} : Luo (Kenya and Tanzania)
{lus} : Lushai
{mk} : Macedonian
eq the modern Slavic language spoken in what was Yugoslavia. NOT the form of Greek spoken in
Greek Macedonia!
{mad} : Madurese
{mag} : Magahi
{mai} : Maithili
{mak} : Makasar
{mg} : Malagasy
{ms} : Malay
NOT Malayalam!
{ml} : Malayalam
NOT Malay!
{mt} : Maltese
{mnc} : Manchu
{mdr} : Mandar
NOT Mandarin!

{man} : Mandingo
{mni} : Manipuri
eq Meithei.

[{mno} : Manobo languages]
{gv} : Manx
{mi} : Maori
NOT Mari!

{mr} : Marathi
{chm} : Mari
NOT Maori!

{mh} : Marshall
eq Marshallese.

{mwr} : Marwari
{mas} : Masai
[{myn} : Mayan languages]
{men} : Mende
{mic} : Micmac
{min} : Minangkabau
{i-mingo} : Mingo
eq the Irquoian language West Virginia Seneca. NOT New York Seneca!

[{mis} : Miscellaneous languages]
Don't use this.

{moh} : Mohawk
{mo} : Moldavian
eq Moldovan.

[{mkh} : Mon-Khmer (Other)]
{lol} : Mongo
{mn} : Mongolian
eq Mongol.

{mos} : Mossi
[{mul} : Multiple languages]
Not for normal use.

[{mun} : Munda languages]
{nah} : Nahuatl
{na} : Nauru
{nv} : Navajo
eq Navaho. (Formerly i-navajo.)
=for etc {i-navajo} Navajo (old tag)

{nd} : North Ndebele
{nr} : South Ndebele
{ng} : Ndonga
{ne} : Nepali
eq Nepalese. Notable forms: {ne-np} Nepal Nepali; {ne-in} India Nepali.

{new} : Newari

{nia} : Nias
[{nic} : Niger–Kordofanian (Other)]
[{ssa} : Nilo–Saharan (Other)]
{niu} : Niuean
{non} : Old Norse
 (Historical)
[{nai} : North American Indian]
 Do not use this.
{se} : Northern Sami
 eq Lappish. eq Lapp. eq (Northern) Saami.
{no} : Norwegian
 Note the two following forms:
{nb} : Norwegian Bokmal
 eq Bokmål, (A form of Norwegian.) (Formerly no–bok.)
 =for etc {no–bok} Norwegian Bokmal (old tag)
{nn} : Norwegian Nynorsk
 (A form of Norwegian.) (Formerly no–nyn.)
 =for etc {no–nyn} Norwegian Nynorsk (old tag)
[{nub} : Nubian languages]
{nym} : Nyamwezi
{nyn} : Nyankole
{nyo} : Nyoro
{nzi} : Nzima
{oc} : Occitan (post 1500)
 eq Provençal, eq Provençal
{oji} : Ojibwa
 eq Ojibwe.
{or} : Oriya
{om} : Oromo
{osa} : Osage
{os} : Ossetian; Ossetic
[{oto} : Otomian languages]
 Group of languages collectively called "Otomí".
{pal} : Pahlavi
 eq Pahlevi
{i–pwn} : Paiwan
 eq Pariwan
{pau} : Palauan
{pi} : Pali
 (Historical?)
{pam} : Pampanga
{pag} : Pangasinan
{pa} : Panjabi
 eq Punjabi

{pap} : Papiamentu
eq Papiamentu.

[{paa} : Papuan (Other)]
{fa} : Persian
eq Farsi. eq Iranian.

{peo} : Old Persian (ca.600–400 B.C.)
[{phi} : Philippine (Other)]
{phn} : Phoenician
(Historical)

{pon} : Pohnpeian
NOT Pompeian!

{pl} : Polish
{pt} : Portuguese
eq Portuguese. Notable forms: {pt-pt} Portugal Portuguese; {pt-br} Brazilian Portuguese.

[{pra} : Prakrit languages]
{pro} : Old Provencal (to 1500)
eq Old Provençal. (Historical.)

{ps} : Pushto
eq Pashto. eq Pushtu.

{qu} : Quechua
eq Quecha.

{rm} : Raeto–Romance
eq Romansh.

{raj} : Rajasthani
{rap} : Rapanui
{rar} : Rarotongan
[{qaa – qtz} : Reserved for local use.]
[{roa} : Romance (Other)]
NOT Romanian! NOT Romany! NOT Romansh!

{ro} : Romanian
eq Rumanian. NOT Romany!

{rom} : Romany
eq Rom. NOT Romanian!

{rn} : Rundi
{ru} : Russian
NOT White Russian! NOT Rusyn!

[{sal} : Salishan languages]
Large language group.

{sam} : Samaritan Aramaic
NOT Aramaic!

[{smi} : Sami languages (Other)]
{sm} : Samoan

{sad} : Sandawe
{sg} : Sango
{sa} : Sanskrit
 (Historical)

{sat} : Santali
{sc} : Sardinian
 eq Sard.

{sas} : Sasak
{sco} : Scots
 NOT Scots Gaelic!

{sel} : Selkup
[sem] : Semitic (Other)
{sr} : Serbian
 eq Serb. NOT Sorbian.

{srr} : Serer
{shn} : Shan
{sn} : Shona
{sid} : Sidamo
{sgn-...} : Sign Languages
 Always use with a subtag. Notable forms: {sgn-gb} British Sign Language (BSL); {sgn-ie} Irish
 Sign Language (ESL); {sgn-ni} Nicaraguan Sign Language (ISN); {sgn-us} American Sign
 Language (ASL).

{bla} : Siksika
 eq Blackfoot. eq Pikanii.

{sd} : Sindhi
{si} : Sinhalese
 eq Sinhala.

[sit] : Sino-Tibetan (Other)
[sio] : Siouan languages
{den} : Slave (Athapaskan)
 ("Slavey" is a subform.)

[sla] : Slavic (Other)
{sk} : Slovak
 eq Slovakian.

{sl} : Slovenian
 eq Slovene.

{sog} : Sogdian
{so} : Somali
{son} : Songhai
{snk} : Soninke
{wen} : Sorbian languages
 eq Wendish. eq Sorb. eq Lusatian. eq Wend. NOT Venda! NOT Serbian!

{nso} : Northern Sotho
{st} : Southern Sotho
 eq Sutu. eq Sesotho.

[{sai} : South American Indian (Other)]

{es} : Spanish

Notable forms: {es-ar} Argentine Spanish; {es-bo} Bolivian Spanish; {es-cl} Chilean Spanish; {es-co} Colombian Spanish; {es-do} Dominican Spanish; {es-ec} Ecuadorian Spanish; {es-es} Spain Spanish; {es-gt} Guatemalan Spanish; {es-hn} Honduran Spanish; {es-mx} Mexican Spanish; {es-pa} Panamanian Spanish; {es-pe} Peruvian Spanish; {es-pr} Puerto Rican Spanish; {es-py} Paraguay Spanish; {es-sv} Salvadoran Spanish; {es-us} US Spanish; {es-uy} Uruguayan Spanish; {es-ve} Venezuelan Spanish.

{suk} : Sukuma

{sux} : Sumerian

(Historical)

{su} : Sundanese

{sus} : Susu

{sw} : Swahili

eq Kiswahili

{ss} : Swati

{sv} : Swedish

Notable forms: {sv-se} Sweden Swedish; {sv-fi} Finland Swedish.

{syr} : Syriac

{tl} : Tagalog

{ty} : Tahitian

[{tai} : Tai (Other)]

NOT Thai!

{tg} : Tajik

{tmh} : Tamashek

{ta} : Tamil

{i-tao} : Tao

eq Yami.

{tt} : Tatar

{i-tay} : Tayal

eq Atayal. eq Atayan.

{te} : Telugu

{ter} : Tereno

{tet} : Tetum

{th} : Thai

NOT Tai!

{bo} : Tibetan

{tig} : Tigre

{ti} : Tigrinya

{tem} : Timne

eq Themne. eq Timene.

{tiv} : Tiv

{tli} : Tlingit

{tpi} : Tok Pisin

{tkl} : Tokelau

{tog} : Tonga (Nyasa)

NOT Tsonga!

{to} : Tonga (Tonga Islands)
(Pronounced "Tong-a", not "Tong-ga")
NOT Tsonga!

{tsi} : Tsimshian
eq Sm'algyax

{ts} : Tsonga
NOT Tonga!

{i-tsu} : Tsou

{tn} : Tswana
Same as Setswana.

{tum} : Tumbuka

{tr} : Turkish
(Typically in Roman script)

{ota} : Ottoman Turkish (1500–1928)
(Typically in Arabic script) (Historical)

{tk} : Turkmen
eq Turkmeni.

{tvl} : Tuvalu

{tyv} : Tuvinian
eq Tuvan. eq Tuvín.

{tw} : Twi

{uga} : Ugaritic
NOT Ugrie!

{ug} : Uighur

{uk} : Ukrainian

{umb} : Umbundu

{und} : Undetermined
Not a tag for normal use.

{ur} : Urdu

{uz} : Uzbek
eq Özbek

{vai} : Vai

{ven} : Venda
NOT Wendish! NOT Wend! NOT Avestan!

{vi} : Vietnamese
eq Viet.

{vo} : Volapük
eq Volapük. (Artificial)

{vot} : Votic
eq Votian. eq Vod.

[{wak} : Wakashan languages]

`{wal}` : Walamo
eq Wolaytta.

`{war}` : Waray
Presumably the Philippine language Waray–Waray (Samareño), not the smaller Philippine language Waray Sorsogon, nor the extinct Australian language Waray.

`{was}` : Washo
eq Washoe

`{cy}` : Welsh
`{wo}` : Wolof
`{x-...}` : Unregistered (Semi–Private Use)
"x–" is a prefix for language tags that are not registered with ISO or IANA. Example, x–double–dutch

`{xh}` : Xhosa
`{sah}` : Yakut
`{yao}` : Yao
(The Yao in Malawi?)

`{yap}` : Yapese
eq Yap

`{yi}` : Yiddish
Formerly "ji". Sometimes in Roman script, sometimes in Hebrew script.
=for etc {ji} Yiddish (old tag)

`{yo}` : Yoruba
`[{ypk}]` : Yupik languages
Several "Eskimo" languages.

`{znd}` : Zande
`[{zap}]` : Zapotec
(A group of languages.)

`{zen}` : Zenaga
NOT Zend.

`{za}` : Zhuang
`{zu}` : Zulu
`{zun}` : Zuni
eq Zuñi
=for woohah END

SEE ALSO

[I18N::LangTags/I18N::LangTags](#) and its "See Also" section.

COPYRIGHT AND DISCLAIMER

Copyright (c) 2001,2002 Sean M. Burke. All rights reserved.

You can redistribute and/or modify this document under the same terms as Perl itself.

This document is provided in the hope that it will be useful, but without any warranty; without even the implied warranty of accuracy, authoritativeness, completeness, merchantability, or fitness for a particular purpose.

Email any corrections or questions to me.

AUTHOR

Sean M. Burke, sburkecpan.org

NAME

if – use a Perl module if a condition holds

SYNOPSIS

```
use if CONDITION, MODULE => ARGUMENTS;
```

DESCRIPTION

The construct

```
use if CONDITION, MODULE => ARGUMENTS;
```

has no effect unless `CONDITION` is true. In this case the effect is the same as of

```
use MODULE ARGUMENTS;
```

BUGS

The current implementation does not allow specification of the required version of the module.

AUTHOR

Ilya Zakharevich <mailto:perl-module-if@ilyaz.org>.

NAME

integer – Perl pragma to use integer arithmetic instead of floating point

SYNOPSIS

```
use integer;
$x = 10/3;
# $x is now 3, not 3.3333333333333333
```

DESCRIPTION

This tells the compiler to use integer operations from here to the end of the enclosing BLOCK. On many machines, this doesn't matter a great deal for most computations, but on those without floating point hardware, it can make a big difference in performance.

Note that this only affects how most of the arithmetic and relational **operators** handle their operands and results, and **not** how all numbers everywhere are treated. Specifically, `use integer;` has the effect that before computing the results of the arithmetic operators (+, −, *, /, %, +=, −=, *=, /=, %=, and unary minus), the comparison operators (<, <=, =, ==, !=, <=), and the bitwise operators (&, ^, <<, >>, &=, ^=, <<=, >>=), the operands have their fractional portions truncated (or floored), and the result will have its fractional portion truncated as well. In addition, the range of operands and results is restricted to that of familiar two's complement integers, i.e., $-(2^{**31}) .. (2^{**31}-1)$ on 32-bit architectures, and $-(2^{**63}) .. (2^{**63}-1)$ on 64-bit architectures. For example, this code

```
use integer;
$x = 5.8;
$y = 2.5;
$z = 2.7;
$a = 2**31 - 1; # Largest positive integer on 32-bit machines
$, = ", ";
print $x, -$x, $x + $y, $x - $y, $x / $y, $x * $y, $y == $z, $a, $a + 1;
```

will print: 5.8, −5, 7, 3, 2, 10, 1, 2147483647, −2147483648

Note that `$x` is still printed as having its true non-integer value of 5.8 since it wasn't operated on. And note too the wrap-around from the largest positive integer to the largest negative one. Also, arguments passed to functions and the values returned by them are **not** affected by `use integer;`. E.g.,

```
srand(1.5);
$, = ", ";
print sin(.5), cos(.5), atan2(1,2), sqrt(2), rand(10);
```

will give the same result with or without `use integer;` The power operator `**` is also not affected, so that `2**.5` is always the square root of 2. Now, it so happens that the pre- and post- increment and decrement operators, `++` and `--`, are not affected by `use integer;` either. Some may rightly consider this to be a bug — but at least it's a long-standing one.

Finally, `use integer;` also has an additional affect on the bitwise operators. Normally, the operands and results are treated as **unsigned** integers, but with `use integer;` the operands and results are **signed**. This means, among other things, that `~0` is `−1`, and `−2 & −5` is `−6`.

Internally, native integer arithmetic (as provided by your C compiler) is used. This means that Perl's own semantics for arithmetic operations may not be preserved. One common source of trouble is the modulus of negative numbers, which Perl does one way, but your hardware may do another.

```
% perl -le 'print (4 % -3)'
-2
% perl -Minteger -le 'print (4 % -3)'
1
```

See [Pragmatic Modules in perlmodlib](#), [Integer Arithmetic in perlop](#)

NAME

IO – load various IO modules

SYNOPSIS

```
use IO;
```

DESCRIPTION

IO provides a simple mechanism to load some of the IO modules at one go. Currently this includes:

```
IO::Handle  
IO::Seekable  
IO::File  
IO::Pipe  
IO::Socket  
IO::Dir
```

For more information on any of these modules, please see its respective documentation.

NAME

IO::Dir – supply object methods for directory handles

SYNOPSIS

```
use IO::Dir;
$d = new IO::Dir ".";
if (defined $d) {
    while (defined($_ = $d->read)) { something($_); }
    $d->rewind;
    while (defined($_ = $d->read)) { something_else($_); }
    undef $d;
}

tie %dir, IO::Dir, ".";
foreach (keys %dir) {
    print $_, " " , $dir{$_}->size, "\n";
}
```

DESCRIPTION

The IO::Dir package provides two interfaces to perl's directory reading routines.

The first interface is an object approach. IO::Dir provides an object constructor and methods, which are just wrappers around perl's built in directory reading routines.

`new ([DIRNAME])`

`new` is the constructor for IO::Dir objects. It accepts one optional argument which, if given, `new` will pass to `open`

The following methods are wrappers for the directory related functions built into perl (the trailing 'dir' has been removed from the names). See [perlfunc](#) for details of these functions.

`open (DIRNAME)`

`read ()`

`seek (POS)`

`tell ()`

`rewind ()`

`close ()`

IO::Dir also provides an interface to reading directories via a tied HASH. The tied HASH extends the interface beyond just the directory reading routines by the use of `lstat`, from the `File::stat` package, `unlink`, `rmdir` and `utime`.

`tie %hash, IO::Dir, DIRNAME [, OPTIONS]`

The keys of the HASH will be the names of the entries in the directory. Reading a value from the hash will be the result of calling `File::stat::lstat`. Deleting an element from the hash will call `unlink` providing that `DIR_UNLINK` is passed in the `OPTIONS`.

Assigning to an entry in the HASH will cause the time stamps of the file to be modified. If the file does not exist then it will be created. Assigning a single integer to a HASH element will cause both the access and modification times to be changed to that value. Alternatively a reference to an array of two values can be passed. The first array element will be used to set the access time and the second element will be used to set the modification time.

SEE ALSO

[File::stat](#)

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs to <perl5-porters@perl.org>.

COPYRIGHT

Copyright (c) 1997–8 Graham Barr <gbarr@pobox.com>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

IO::File – supply object methods for filehandles

SYNOPSIS

```
use IO::File;

$fh = new IO::File;
if ($fh->open("< file")) {
    print <$fh>;
    $fh->close;
}

$fh = new IO::File "> file";
if (defined $fh) {
    print $fh "bar\n";
    $fh->close;
}

$fh = new IO::File "file", "r";
if (defined $fh) {
    print <$fh>;
    undef $fh;          # automatically closes the file
}

$fh = new IO::File "file", O_WRONLY|O_APPEND;
if (defined $fh) {
    print $fh "corge\n";

    $pos = $fh->getpos;
    $fh->setpos($pos);

    undef $fh;          # automatically closes the file
}

autoflush STDOUT 1;
```

DESCRIPTION

IO::File inherits from IO::Handle and IO::Seekable. It extends these classes with methods that are specific to file handles.

CONSTRUCTOR

`new (FILENAME [,MODE [,PERMS]])`

Creates an IO::File. If it receives any parameters, they are passed to the method `open`; if the open fails, the object is destroyed. Otherwise, it is returned to the caller.

`new_tmpfile`

Creates an IO::File opened for read/write on a newly created temporary file. On systems where this is possible, the temporary file is anonymous (i.e. it is unlinked after creation, but held open). If the temporary file cannot be created or opened, the IO::File object is destroyed. Otherwise, it is returned to the caller.

METHODS

`open(FILENAME [,MODE [,PERMS]])`

`open` accepts one, two or three parameters. With one parameter, it is just a front end for the built-in `open` function. With two or three parameters, the first parameter is a filename that may include whitespace or other special characters, and the second parameter is the open mode, optionally followed by a file permission value.

If `IO::File::open` receives a Perl mode string ("`>`", "`+<`", etc.) or an ANSI C `fopen()` mode string ("`w`", "`r+`", etc.), it uses the basic Perl `open` operator (but protects any special characters).

If `IO::File::open` is given a numeric mode, it passes that mode and the optional permissions value to the Perl `sysopen` operator. The permissions default to 0666.

For convenience, `IO::File` exports the `O_XXX` constants from the `Fcntl` module, if this module is available.

SEE ALSO

[perlfunc](#), [I/O Operators in perlop](#), [IO::Handle](#), [IO::Seekable](#)

HISTORY

Derived from `FileHandle.pm` by Graham Barr <gbarr@pobox.com>.

NAME

IO::Handle – supply object methods for I/O handles

SYNOPSIS

```
use IO::Handle;

$io = new IO::Handle;
if ($io->fdopen(fileno(STDIN), "r")) {
    print $io->getline;
    $io->close;
}

$io = new IO::Handle;
if ($io->fdopen(fileno(STDOUT), "w")) {
    $io->print("Some text\n");
}

use IO::Handle '_IOLBF';
$io->setvbuf($buffer_var, _IOLBF, 1024);

undef $io;          # automatically closes the file if it's open

autoflush STDOUT 1;
```

DESCRIPTION

IO::Handle is the base class for all other IO handle classes. It is not intended that objects of IO::Handle would be created directly, but instead IO::Handle is inherited from by several other classes in the IO hierarchy.

If you are reading this documentation, looking for a replacement for the FileHandle package, then I suggest you read the documentation for IO::File too.

CONSTRUCTOR

`new ()`

Creates a new IO::Handle object.

`new_from_fd (FD, MODE)`

Creates an IO::Handle like new does. It requires two parameters, which are passed to the method `fdopen`; if the `fdopen` fails, the object is destroyed. Otherwise, it is returned to the caller.

METHODS

See [perlfunc](#) for complete descriptions of each of the following supported IO::Handle methods, which are just front ends for the corresponding built-in functions:

```
$io->close
$io->eof
$io->fileno
$io->format_write( [FORMAT_NAME] )
$io->getc
$io->read ( BUF, LEN, [OFFSET] )
$io->print ( ARGS )
$io->printf ( FMT, [ARGS] )
$io->stat
$io->sysread ( BUF, LEN, [OFFSET] )
$io->syswrite ( BUF, [LEN, [OFFSET]] )
$io->truncate ( LEN )
```

See [perlvar](#) for complete descriptions of each of the following supported IO::Handle methods. All of them return the previous value of the attribute and takes an optional single argument that when given will set

the value. If no argument is given the previous value is unchanged (except for `$io->autoflush` will actually turn ON autoflush by default).

```

$io->autoflush ( [BOOL] )           $|
$io->format_page_number( [NUM] )     $%
$io->format_lines_per_page( [NUM] )  $=
$io->format_lines_left( [NUM] )      $-
$io->format_name( [STR] )            $~
$io->format_top_name( [STR] )        $^
$io->input_line_number( [NUM] )      $.

```

The following methods are not supported on a per-filehandle basis.

```

IO::Handle->format_line_break_characters( [STR] ) $:
IO::Handle->format_formfeed( [STR] )             $^L
IO::Handle->output_field_separator( [STR] )       $,
IO::Handle->output_record_separator( [STR] )      $\
IO::Handle->input_record_separator( [STR] )        $/

```

Furthermore, for doing normal I/O you might need these:

`$io->fdopen (FD, MODE)`

`fdopen` is like an ordinary `open` except that its first parameter is not a filename but rather a file handle name, an `IO::Handle` object, or a file descriptor number.

`$io->opened`

Returns true if the object is currently a valid file descriptor, false otherwise.

`$io->getline`

This works like `<$io` described in *[I/O Operators in perlop](#)* except that it's more readable and can be safely called in a list context but still returns just one line.

`$io->getlines`

This works like `<$io` when called in a list context to read all the remaining lines in a file, except that it's more readable. It will also `croak()` if accidentally called in a scalar context.

`$io->ungetc (ORD)`

Pushes a character with the given ordinal value back onto the given handle's input stream. Only one character of pushback per handle is guaranteed.

`$io->write (BUF, LEN [, OFFSET])`

This `write` is like `write` found in C, that is it is the opposite of `read`. The wrapper for the perl `write` function is called `format_write`.

`$io->error`

Returns a true value if the given handle has experienced any errors since it was opened or since the last call to `clearerr`, or if the handle is invalid. It only returns false for a valid handle with no outstanding errors.

`$io->clearerr`

Clear the given handle's error indicator. Returns `-1` if the handle is invalid, `0` otherwise.

`$io->sync`

`sync` synchronizes a file's in-memory state with that on the physical medium. `sync` does not operate at the `perlio` api level, but operates on the file descriptor (similar to `sysread`, `sysseek` and `system`). This means that any data held at the `perlio` api level will not be synchronized. To synchronize data that is buffered at the `perlio` api level you must use the `flush` method. `sync` is not implemented on all platforms. Returns "0 but true" on success, `undef` on error, `undef` for an invalid handle. See *[fsync\(3c\)](#)*.

`$io-flush`

`flush` causes perl to flush any buffered data at the `perlio` api level. Any unread data in the buffer will be discarded, and any unwritten data will be written to the underlying file descriptor. Returns "0 but true" on success, `undef` on error.

`$io-printflush (ARGS)`

Turns on autoflush, print `ARGS` and then restores the autoflush status of the `IO::Handle` object. Returns the return value from `print`.

`$io-blocking ([BOOL])`

If called with an argument `blocking` will turn on non-blocking IO if `BOOL` is false, and turn it off if `BOOL` is true.

`blocking` will return the value of the previous setting, or the current setting if `BOOL` is not given.

If an error occurs `blocking` will return `undef` and `$!` will be set.

If the C functions `setbuf()` and/or `setvbuf()` are available, then `IO::Handle::setbuf` and `IO::Handle::setvbuf` set the buffering policy for an `IO::Handle`. The calling sequences for the Perl functions are the same as their C counterparts—including the constants `_IOFBF`, `_IOLBF`, and `_IONBF` for `setvbuf()`—except that the buffer parameter specifies a scalar variable to use as a buffer. You should only change the buffer before any I/O, or immediately after calling `flush`.

WARNING: A variable used as a buffer by `setbuf` or `setvbuf` **must not be modified** in any way until the `IO::Handle` is closed or `setbuf` or `setvbuf` is called again, or memory corruption may result! Remember that the order of global destruction is undefined, so even if your buffer variable remains in scope until program termination, it may be undefined before the file `IO::Handle` is closed. Note that you need to import the constants `_IOFBF`, `_IOLBF`, and `_IONBF` explicitly. Like C, `setbuf` returns nothing. `setvbuf` returns "0 but true", on success, `undef` on failure.

Lastly, there is a special method for working under `-T` and `setuid/gid` scripts:

`$io-untaint`

Marks the object as taint-clean, and as such data read from it will also be considered taint-clean. Note that this is a very trusting action to take, and appropriate consideration for the data source and potential vulnerability should be kept in mind. Returns 0 on success, -1 if setting the taint-clean flag failed. (eg invalid handle)

NOTE

An `IO::Handle` object is a reference to a symbol/GLOB reference (see the `Symbol` package). Some modules that inherit from `IO::Handle` may want to keep object related variables in the hash table part of the GLOB. In an attempt to prevent modules trampling on each other I propose that any such module should prefix its variables with its own name separated by `_`'s. For example the `IO::Socket` module keeps a `timeout` variable in `'io_socket_timeout'`.

SEE ALSO

[perlfunc](#), [I/O Operators in perlop](#), [IO::File](#)

BUGS

Due to backwards compatibility, all filehandles resemble objects of class `IO::Handle`, or actually classes derived from that class. They actually aren't. Which means you can't derive your own class from `IO::Handle` and inherit those methods.

HISTORY

Derived from `FileHandle.pm` by Graham Barr <gbarr@pobox.com>

NAME

IO::Pipe – supply object methods for pipes

SYNOPSIS

```
use IO::Pipe;

$pipe = new IO::Pipe;

if($pid = fork()) { # Parent
    $pipe->reader();

    while(<$pipe>) {
        ...
    }
}

elsif(defined $pid) { # Child
    $pipe->writer();

    print $pipe ...
}

or

$pipe = new IO::Pipe;
$pipe->reader(qw(ls -l));

while(<$pipe>) {
    ...
}
```

DESCRIPTION

IO::Pipe provides an interface to creating pipes between processes.

CONSTRUCTOR

`new ([READER, WRITER])`

Creates an `IO::Pipe`, which is a reference to a newly created symbol (see the `Symbol` package). `IO::Pipe::new` optionally takes two arguments, which should be objects blessed into `IO::Handle`, or a subclass thereof. These two objects will be used for the system call to `pipe`. If no arguments are given then `method handles` is called on the new `IO::Pipe` object.

These two handles are held in the array part of the GLOB until either `reader` or `writer` is called.

METHODS

`reader ([ARGS])`

The object is re-blessed into a sub-class of `IO::Handle`, and becomes a handle at the reading end of the pipe. If `ARGS` are given then `fork` is called and `ARGS` are passed to `exec`.

`writer ([ARGS])`

The object is re-blessed into a sub-class of `IO::Handle`, and becomes a handle at the writing end of the pipe. If `ARGS` are given then `fork` is called and `ARGS` are passed to `exec`.

`handles ()`

This method is called during construction by `IO::Pipe::new` on the newly created `IO::Pipe` object. It returns an array of two objects blessed into `IO::Pipe::End`, or a subclass thereof.

SEE ALSO

[IO::Handle](#)

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs to <perl5-porters@perl.org>.

COPYRIGHT

Copyright (c) 1996–8 Graham Barr <gbarr@pobox.com>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

IO::Poll – Object interface to system poll call

SYNOPSIS

```
use IO::Poll qw(POLLRDNORM POLLWRNORM POLLIN POLLHUP);

$poll = new IO::Poll;

$poll->mask($input_handle => POLLIN);
$poll->mask($output_handle => POLLOUT);

$poll->poll($timeout);

$ev = $poll->events($input);
```

DESCRIPTION

IO::Poll is a simple interface to the system level poll routine.

METHODS

mask (IO [, EVENT_MASK])

If EVENT_MASK is given, then, if EVENT_MASK is non-zero, IO is added to the list of file descriptors and the next call to poll will check for any event specified in EVENT_MASK. If EVENT_MASK is zero then IO will be removed from the list of file descriptors.

If EVENT_MASK is not given then the return value will be the current event mask value for IO.

poll ([TIMEOUT])

Call the system level poll routine. If TIMEOUT is not specified then the call will block. Returns the number of handles which had events happen, or -1 on error.

events (IO)

Returns the event mask which represents the events that happen on IO during the last call to poll.

remove (IO)

Remove IO from the list of file descriptors for the next poll.

handles([EVENT_MASK])

Returns a list of handles. If EVENT_MASK is not given then a list of all handles known will be returned. If EVENT_MASK is given then a list of handles will be returned which had one of the events specified by EVENT_MASK happen during the last call to poll.

SEE ALSO

[*poll\(2\)*](#), [*IO::Handle*](#), [*IO::Select*](#)

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs to <perl5-porters@perl.org>.

COPYRIGHT

Copyright (c) 1997–8 Graham Barr <gbarr@pobox.com>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

IO::Seekable – supply seek based methods for I/O objects

SYNOPSIS

```
use IO::Seekable;
package IO::Something;
@ISA = qw(IO::Seekable);
```

DESCRIPTION

IO::Seekable does not have a constructor of its own as it is intended to be inherited by other IO::Handle based objects. It provides methods which allow seeking of the file descriptors.

\$io-getpos

Returns an opaque value that represents the current position of the IO::File, or undef if this is not possible (eg an unseekable stream such as a terminal, pipe or socket). If the `fgetpos()` function is available in your C library it is used to implements `getpos`, else perl emulates `getpos` using C's `ftell()` function.

\$io-setpos

Uses the value of a previous `getpos` call to return to a previously visited position. Returns "0 but true" on success, undef on failure.

See [perlfunc](#) for complete descriptions of each of the following supported IO::Seekable methods, which are just front ends for the corresponding built-in functions:

\$io-seek (POS, WHENCE)

Seek the IO::File to position POS, relative to WHENCE:

WHENCE=0 (SEEK_SET)

POS is absolute position. (Seek relative to the start of the file)

WHENCE=1 (SEEK_CUR)

POS is an offset from the current position. (Seek relative to current)

WHENCE=2 (SEEK_END)

POS is an offset from the end of the file. (Seek relative to end)

The SEEK_* constants can be imported from the `Fcntl` module if you don't wish to use the numbers 1 or 2 in your code.

Returns 1 upon success, otherwise.

\$io-sysseek(POS, WHENCE)

Similar to `$io-seek`, but sets the IO::File's position using the system call `lseek(2)` directly, so will confuse most perl IO operators except `sysread` and `syswrite` (see [perlfunc](#) for full details)

Returns the new position, or undef on failure. A position of zero is returned as the string "0 but true"

\$io-tell

Returns the IO::File's current position, or -1 on error.

SEE ALSO

[perlfunc](#), *I/O Operators in perlop*, *IO::Handle* *IO::File*

HISTORY

Derived from FileHandle.pm by Graham Barr <gbarr@pobox.com>

NAME

IO::Select – OO interface to the select system call

SYNOPSIS

```
use IO::Select;

$s = IO::Select->new();

$s->add(\*STDIN);
$s->add($some_handle);

@ready = $s->can_read($timeout);

@ready = IO::Select->new(@handles)->read(0);
```

DESCRIPTION

The `IO::Select` package implements an object approach to the system `select` function call. It allows the user to see what IO handles, see [IO::Handle](#), are ready for reading, writing or have an error condition pending.

CONSTRUCTOR

`new ([HANDLES])`

The constructor creates a new object and optionally initialises it with a set of handles.

METHODS

`add (HANDLES)`

Add the list of handles to the `IO::Select` object. It is these values that will be returned when an event occurs. `IO::Select` keeps these values in a cache which is indexed by the `fileno` of the handle, so if more than one handle with the same `fileno` is specified then only the last one is cached.

Each handle can be an `IO::Handle` object, an integer or an array reference where the first element is an `IO::Handle` or an integer.

`remove (HANDLES)`

Remove all the given handles from the object. This method also works by the `fileno` of the handles. So the exact handles that were added need not be passed, just handles that have an equivalent `fileno`

`exists (HANDLE)`

Returns a true value (actually the handle itself) if it is present. Returns `undef` otherwise.

`handles`

Return an array of all registered handles.

`can_read ([TIMEOUT])`

Return an array of handles that are ready for reading. `TIMEOUT` is the maximum amount of time to wait before returning an empty list, in seconds, possibly fractional. If `TIMEOUT` is not given and any handles are registered then the call will block.

`can_write ([TIMEOUT])`

Same as `can_read` except check for handles that can be written to.

`has_exception ([TIMEOUT])`

Same as `can_read` except check for handles that have an exception condition, for example pending out-of-band data.

`count ()`

Returns the number of handles that the object will check for when one of the `can_` methods is called or the object is passed to the `select` static method.

bits()

Return the bit string suitable as argument to the core `select()` call.

select (READ, WRITE, ERROR [, TIMEOUT])

`select` is a static method, that is you call it with the package name like `new`. `READ`, `WRITE` and `ERROR` are either `undef` or `IO::Select` objects. `TIMEOUT` is optional and has the same effect as for the core `select` call.

The result will be an array of 3 elements, each a reference to an array which will hold the handles that are ready for reading, writing and have error conditions respectively. Upon error an empty array is returned.

EXAMPLE

Here is a short example which shows how `IO::Select` could be used to write a server which communicates with several sockets while also listening for more connections on a listen socket

```
use IO::Select;
use IO::Socket;

$lsn = new IO::Socket::INET(Listen => 1, LocalPort => 8080);
$sel = new IO::Select( $lsn );

while(@ready = $sel->can_read) {
    foreach $fh (@ready) {
        if($fh == $lsn) {
            # Create a new socket
            $new = $lsn->accept;
            $sel->add($new);
        }
        else {
            # Process socket

            # Maybe we have finished with the socket
            $sel->remove($fh);
            $fh->close;
        }
    }
}
```

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs to <perl5-porters@perl.org>.

COPYRIGHT

Copyright (c) 1997-8 Graham Barr <gbarr@pobox.com>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

IO::Socket – Object interface to socket communications

SYNOPSIS

```
use IO::Socket;
```

DESCRIPTION

IO::Socket provides an object interface to creating and using sockets. It is built upon the [IO::Handle](#) interface and inherits all the methods defined by [IO::Handle](#).

IO::Socket only defines methods for those operations which are common to all types of socket. Operations which are specified to a socket in a particular domain have methods defined in sub classes of IO::Socket

IO::Socket will export all functions (and constants) defined by [Socket](#).

CONSTRUCTOR

```
new ( [ARGS] )
```

Creates an IO::Socket, which is a reference to a newly created symbol (see the Symbol package). new optionally takes arguments, these arguments are in key–value pairs. new only looks for one key Domain which tells new which domain the socket will be in. All other arguments will be passed to the configuration method of the package for that domain, See below.

NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE

As of VERSION 1.18 all IO::Socket objects have autoflush turned on by default. This was not the case with earlier releases.

NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE

METHODS

See [perlfunc](#) for complete descriptions of each of the following supported IO::Socket methods, which are just front ends for the corresponding built–in functions:

```
socket
socketpair
bind
listen
accept
send
recv
peername (getpeername)
sockname (getsockname)
shutdown
```

Some methods take slightly different arguments to those defined in [perlfunc](#) in attempt to make the interface more flexible. These are

```
accept([PKG])
```

perform the system call `accept` on the socket and return a new object. The new object will be created in the same class as the listen socket, unless PKG is specified. This object can be used to communicate with the client that was trying to connect.

In a scalar context the new socket is returned, or undef upon failure. In a list context a two–element array is returned containing the new socket and the peer address; the list will be empty upon failure.

The timeout in the [PKG] can be specified as zero to effect a "poll", but you shouldn't do that because a new IO::Select object will be created behind the scenes just to do the single poll. This is horrendously inefficient. Use rather `true select()` with a zero timeout on the handle, or

non-blocking IO.

socketpair(DOMAIN, TYPE, PROTOCOL)

Call `socketpair` and return a list of two sockets created, or an empty list on failure.

Additional methods that are provided are:

atmark

True if the socket is currently positioned at the urgent data mark, false otherwise.

```
use IO::Socket;

my $sock = IO::Socket::INET->new('some_server');
$sock->read(1024,$data) until $sock->atmark;
```

Note: this is a reasonably new addition to the family of socket functions, so all systems may not support this yet. If it is unsupported by the system, an attempt to use this method will abort the program.

The `atmark()` functionality is also exportable as `socketatmark()` function:

```
use IO::Socket 'socketatmark';
```

This allows for a more traditional use of `socketatmark()` as a procedural socket function. If your system does not support `socketatmark()`, the use declaration will fail at compile time.

connected

If the socket is in a connected state the peer address is returned. If the socket is not in a connected state then `undef` will be returned.

protocol

Returns the numerical number for the protocol being used on the socket, if known. If the protocol is unknown, as with an `AF_UNIX` socket, zero is returned.

sockdomain

Returns the numerical number for the socket domain type. For example, for an `AF_INET` socket the value of `&AF_INET` will be returned.

sockopt(OPT [, VAL])

Unified method to both set and get options in the `SOL_SOCKET` level. If called with one argument then `getsockopt` is called, otherwise `setsockopt` is called.

socktype

Returns the numerical number for the socket type. For example, for a `SOCK_STREAM` socket the value of `&SOCK_STREAM` will be returned.

timeout([VAL])

Set or get the timeout value associated with this socket. If called without any arguments then the current setting is returned. If called with an argument the current setting is changed and the previous value returned.

SEE ALSO

Socket, *IO::Handle*, *IO::Socket::INET*, *IO::Socket::UNIX*

AUTHOR

Graham Barr. `atmark()` by Lincoln Stein. Currently maintained by the Perl Porters. Please report all bugs to perl5-porters@perl.org.

COPYRIGHT

Copyright (c) 1997–8 Graham Barr <gbarr@pobox.com>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

The `atmark()` implementation: Copyright 2001, Lincoln Stein <lstein@cshl.org. This module is distributed under the same terms as Perl itself. Feel free to use, modify and redistribute it as long as you retain the correct attribution.

NAME

IO::Socket::INET – Object interface for AF_INET domain sockets

SYNOPSIS

```
use IO::Socket::INET;
```

DESCRIPTION

IO::Socket::INET provides an object interface to creating and using sockets in the AF_INET domain. It is built upon the *IO::Socket* interface and inherits all the methods defined by *IO::Socket*.

CONSTRUCTOR

```
new ( [ARGS] )
```

Creates an IO::Socket::INET object, which is a reference to a newly created symbol (see the Symbol package). new optionally takes arguments, these arguments are in key–value pairs.

In addition to the key–value pairs accepted by *IO::Socket*, IO::Socket::INET provides.

PeerAddr	Remote host address	<hostname>[:<port>]
PeerHost	Synonym for PeerAddr	
PeerPort	Remote port or service	<service>[(<no>)] <no>
LocalAddr	Local host bind address	hostname[:port]
LocalHost	Synonym for LocalAddr	
LocalPort	Local host bind port	<service>[(<no>)] <no>
Proto	Protocol name (or number)	"tcp" "udp" ...
Type	Socket type	SOCK_STREAM SOCK_DGRAM ...
Listen	Queue size for listen	
ReuseAddr	Set SO_REUSEADDR before binding	
Reuse	Set SO_REUSEADDR before binding (deprecated, prefer ReuseAddr)	
ReusePort	Set SO_REUSEPORT before binding	
Broadcast	Set SO_BROADCAST before binding	
Timeout	Timeout value for various operations	
MultiHomed	Try all addresses for multi-homed hosts	
Blocking	Determine if connection will be blocking mode	

If Listen is defined then a listen socket is created, else if the socket type, which is derived from the protocol, is SOCK_STREAM then connect () is called.

Although it is not illegal, the use of MultiHomed on a socket which is in non–blocking mode is of little use. This is because the first connect will never fail with a timeout as the connect call will not block.

The PeerAddr can be a hostname or the IP–address on the "xx.xx.xx.xx" form. The PeerPort can be a number or a symbolic service name. The service name might be followed by a number in parenthesis which is used if the service is not known by the system. The PeerPort specification can also be embedded in the PeerAddr by preceding it with a ":".

If Proto is not given and you specify a symbolic PeerPort port, then the constructor will try to derive Proto from the service name. As a last resort Proto "tcp" is assumed. The Type parameter will be deduced from Proto if not specified.

If the constructor is only passed a single argument, it is assumed to be a PeerAddr specification.

If Blocking is set to 0, the connection will be in nonblocking mode. If not specified it defaults to 1 (blocking mode).

Examples:

```
$sock = IO::Socket::INET->new (PeerAddr => 'www.perl.org',
                               PeerPort => 'http(80)',
```



```

                                Proto    => 'tcp');
$sock = IO::Socket::INET->new(PeerAddr => 'localhost:smtp(25)');
$sock = IO::Socket::INET->new(Listen    => 5,
                                LocalAddr => 'localhost',
                                LocalPort => 9000,
                                Proto     => 'tcp');

$sock = IO::Socket::INET->new('127.0.0.1:25');
$sock = IO::Socket::INET->new(PeerPort  => 9999,
                                PeerAddr  => inet_ntoa(INADDR_BROADCAST),
                                Proto     => udp,
                                LocalAddr => 'localhost',
                                Broadcast => 1 )
                                or die "Can't bind : $@\n";

```

NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE

As of VERSION 1.18 all IO::Socket objects have autoflush turned on by default. This was not the case with earlier releases.

NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE

METHODS

`sockaddr ()`

Return the address part of the sockaddr structure for the socket

`sockport ()`

Return the port number that the socket is using on the local host

`sockhost ()`

Return the address part of the sockaddr structure for the socket in a text form xx.xx.xx.xx

`peeraddr ()`

Return the address part of the sockaddr structure for the socket on the peer host

`peerport ()`

Return the port number for the socket on the peer host.

`peerhost ()`

Return the address part of the sockaddr structure for the socket on the peer host in a text form xx.xx.xx.xx

SEE ALSO

Socket, *IO::Socket*

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs to <perl5-porters@perl.org>.

COPYRIGHT

Copyright (c) 1996–8 Graham Barr <gbarr@pobox.com>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

IO::Socket::UNIX – Object interface for AF_UNIX domain sockets

SYNOPSIS

```
use IO::Socket::UNIX;
```

DESCRIPTION

IO::Socket::UNIX provides an object interface to creating and using sockets in the AF_UNIX domain. It is built upon the *IO::Socket* interface and inherits all the methods defined by *IO::Socket*.

CONSTRUCTOR

```
new ( [ARGS] )
```

Creates an IO::Socket::UNIX object, which is a reference to a newly created symbol (see the Symbol package). new optionally takes arguments, these arguments are in key–value pairs.

In addition to the key–value pairs accepted by *IO::Socket*, IO::Socket::UNIX provides.

Type	Type of socket (eg SOCK_STREAM or SOCK_DGRAM)
Local	Path to local fifo
Peer	Path to peer fifo
Listen	Create a listen socket

If the constructor is only passed a single argument, it is assumed to be a Peer specification.

NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE

As of VERSION 1.18 all IO::Socket objects have autoflush turned on by default. This was not the case with earlier releases.

NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE NOTE

METHODS

```
hostpath()
```

Returns the pathname to the fifo at the local end

```
peerpath()
```

Returns the pathanme to the fifo at the peer end

SEE ALSO

Socket, *IO::Socket*

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs to <perl5-porters@perl.org>.

COPYRIGHT

Copyright (c) 1996–8 Graham Barr <gbarr@pobox.com>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

IPC::Open2, open2 – open a process for both reading and writing

SYNOPSIS

```
use IPC::Open2;

$pid = open2(\*RDRFH, \*WTRFH, 'some cmd and args');
# or without using the shell
$pid = open2(\*RDRFH, \*WTRFH, 'some', 'cmd', 'and', 'args');

# or with handle autovivification
my($rdrfh, $wtrfh);
$pid = open2($rdrfh, $wtrfh, 'some cmd and args');
# or without using the shell
$pid = open2($rdrfh, $wtrfh, 'some', 'cmd', 'and', 'args');
```

DESCRIPTION

The `open2()` function runs the given `$cmd` and connects `$rdrfh` for reading and `$wtrfh` for writing. It's what you think should work when you try

```
$pid = open(HANDLE, "|cmd args|");
```

The write filehandle will have autoflush turned on.

If `$rdrfh` is a string (that is, a bareword filehandle rather than a glob or a reference) and it begins with `< &`, then the child will send output directly to that file handle. If `$wtrfh` is a string that begins with `< <&`, then `$wtrfh` will be closed in the parent, and the child will read from it directly. In both cases, there will be a `dup(2)` instead of a `pipe(2)` made.

If either reader or writer is the null string, this will be replaced by an autogenerated filehandle. If so, you must pass a valid `lvalue` in the parameter slot so it can be overwritten in the caller, or an exception will be raised.

`open2()` returns the process ID of the child process. It doesn't return on failure: it just raises an exception matching `^open2:/.` . However, `exec` failures in the child are not detected. You'll have to trap `SIGPIPE` yourself.

`open2()` does not wait for and reap the child process after it exits. Except for short programs where it's acceptable to let the operating system take care of this, you need to do this yourself. This is normally as simple as calling `waitpid $pid, 0` when you're done with the process. Failing to do this can result in an accumulation of defunct or "zombie" processes. See [waitpid](#) for more information.

This whole affair is quite dangerous, as you may block forever. It assumes it's going to talk to something like `bc`, both writing to it and reading from it. This is presumably safe because you "know" that commands like `bc` will read a line at a time and output a line at a time. Programs like `sort` that read their entire input stream first, however, are quite apt to cause deadlock.

The big problem with this approach is that if you don't have control over source code being run in the child process, you can't control what it does with pipe buffering. Thus you can't just open a pipe to `cat -v` and continually read and write a line from it.

The `IO::Pty` and `Expect` modules from CPAN can help with this, as they provide a real tty (well, a pseudo-tty, actually), which gets you back to line buffering in the invoked command again.

WARNING

The order of arguments differs from that of `open3()`.

SEE ALSO

See [IPC::Open3](#) for an alternative that handles `STDERR` as well. This function is really just a wrapper around `open3()`.

NAME

IPC::Open3, open3 – open a process for reading, writing, and error handling

SYNOPSIS

```
$pid = open3(\*WTRFH, \*RDRFH, \*ERRFH,  
            'some cmd and args', 'optarg', ...);  
  
my($wtr, $rdr, $err);  
$pid = open3($wtr, $rdr, $err,  
            'some cmd and args', 'optarg', ...);
```

DESCRIPTION

Extremely similar to `open2()`, `open3()` spawns the given `$cmd` and connects `RDRFH` for reading, `WTRFH` for writing, and `ERRFH` for errors. If `ERRFH` is false, or the same file descriptor as `RDRFH`, then `STDOUT` and `STDERR` of the child are on the same filehandle. The `WTRFH` will have autoflush turned on.

If `WTRFH` begins with `< &`, then `WTRFH` will be closed in the parent, and the child will read from it directly. If `RDRFH` or `ERRFH` begins with `< &`, then the child will send output directly to that filehandle. In both cases, there will be a `dup(2)` instead of a `pipe(2)` made.

If either reader or writer is the null string, this will be replaced by an autogenerated filehandle. If so, you must pass a valid lvalue in the parameter slot so it can be overwritten in the caller, or an exception will be raised.

The filehandles may also be integers, in which case they are understood as file descriptors.

`open3()` returns the process ID of the child process. It doesn't return on failure: it just raises an exception matching `/^open3:/. However, exec failures in the child are not detected. You'll have to trap SIGPIPE yourself.`

`open3()` does not wait for and reap the child process after it exits. Except for short programs where it's acceptable to let the operating system take care of this, you need to do this yourself. This is normally as simple as calling `waitpid $pid, 0` when you're done with the process. Failing to do this can result in an accumulation of defunct or "zombie" processes. See [waitpid](#) for more information.

If you try to read from the child's stdout writer and their stderr writer, you'll have problems with blocking, which means you'll want to use `select()` or the `IO::Select`, which means you'd best use `sysread()` instead of `readline()` for normal stuff.

This is very dangerous, as you may block forever. It assumes it's going to talk to something like **bc**, both writing to it and reading from it. This is presumably safe because you "know" that commands like **bc** will read a line at a time and output a line at a time. Programs like **sort** that read their entire input stream first, however, are quite apt to cause deadlock.

The big problem with this approach is that if you don't have control over source code being run in the child process, you can't control what it does with pipe buffering. Thus you can't just open a pipe to `cat -v` and continually read and write a line from it.

WARNING

The order of arguments differs from that of `open2()`.

NAME

IPC::Msg – SysV Msg IPC object class

SYNOPSIS

```
use IPC::SysV qw(IPC_PRIVATE S_IRWXU);
use IPC::Msg;

$msg = new IPC::Msg(IPC_PRIVATE, S_IRWXU);
$msg->snd(pack("l! a*", $msgtype, $msg));
$msg->rcv($buf, 256);

$ds = $msg->stat;

$msg->remove;
```

DESCRIPTION

A class providing an object based interface to SysV IPC message queues.

METHODS

new (KEY , FLAGS)

Creates a new message queue associated with KEY. A new queue is created if

- KEY is equal to IPC_PRIVATE
- KEY does not already have a message queue associated with it, and *FLAGS* & IPC_CREAT is true.

On creation of a new message queue *FLAGS* is used to set the permissions.

id Returns the system message queue identifier.

rcv (BUF, LEN [, TYPE [, FLAGS]])

Read a message from the queue. Returns the type of the message read. See [msgrcv](#). The BUF becomes tainted.

remove

Remove and destroy the message queue from the system.

set (STAT)

set (NAME = VALUE [, NAME = VALUE ...])

set will set the following values of the stat structure associated with the message queue.

```
uid
gid
mode (oly the permission bits)
qbytes
```

set accepts either a stat object, as returned by the stat method, or a list of *name–value* pairs.

snd (TYPE, MSG [, FLAGS])

Place a message on the queue with the data from MSG and with type TYPE. See [msgsnd](#).

stat Returns an object of type IPC::Msg::stat which is a sub-class of Class::Struct. It provides the following fields. For a description of these fields see you system documentation.

```
uid
gid
cuid
cgid
mode
```

qnum
qbytes
lspid
lrpid
stime
rtime
ctime

SEE ALSO

[IPC::SysV Class::Struct](#)

AUTHOR

Graham Barr <gbarr@pobox.com>

COPYRIGHT

Copyright (c) 1997 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

IPC::Semaphore – SysV Semaphore IPC object class

SYNOPSIS

```
use IPC::SysV qw(IPC_PRIVATE S_IRWXU IPC_CREAT);
use IPC::Semaphore;

$sem = new IPC::Semaphore(IPC_PRIVATE, 10, S_IRWXU | IPC_CREAT);
$sem->setall( (0) x 10);
@sem = $sem->getall;
$ncnt = $sem->getncnt;
$zcnt = $sem->getzcnt;
$ds = $sem->stat;
$sem->remove;
```

DESCRIPTION

A class providing an object based interface to SysV IPC semaphores.

METHODS

new (KEY , NSEMS , FLAGS)

Create a new semaphore set associated with KEY. NSEMS is the number of semaphores in the set. A new set is created if

- KEY is equal to IPC_PRIVATE
- KEY does not already have a semaphore identifier associated with it, and *FLAGS* & IPC_CREAT is true.

On creation of a new semaphore set *FLAGS* is used to set the permissions.

getall

Returns the values of the semaphore set as an array.

getncnt (SEM)

Returns the number of processes waiting for the semaphore SEM to become greater than its current value

getpid (SEM)

Returns the process id of the last process that performed an operation on the semaphore SEM.

getval (SEM)

Returns the current value of the semaphore SEM.

getzcnt (SEM)

Returns the number of processes waiting for the semaphore SEM to become zero.

id Returns the system identifier for the semaphore set.

op (OPLIST)

OPLIST is a list of operations to pass to *semop*. OPLIST is a concatenation of smaller lists, each which has three values. The first is the semaphore number, the second is the operation and the last is a flags value. See [semop](#) for more details. For example

```
$sem->op(
    0, -1, IPC_NOWAIT,
    1,  1, IPC_NOWAIT
```

```
) ;
```

remove

Remove and destroy the semaphore set from the system.

set (STAT)

set (NAME = VALUE [, NAME = VALUE ...])

set will set the following values of the stat structure associated with the semaphore set.

```
uid
gid
mode (only the permission bits)
```

set accepts either a stat object, as returned by the stat method, or a list of *name-value* pairs.

setall (VALUES)

Sets all values in the semaphore set to those given on the VALUES list. VALUES must contain the correct number of values.

setval (N , VALUE)

Set the Nth value in the semaphore set to VALUE

stat Returns an object of type `IPC::Semaphore::stat` which is a sub-class of `Class::Struct`. It provides the following fields. For a description of these fields see your system documentation.

```
uid
gid
cuid
cgid
mode
ctime
otime
nsems
```

SEE ALSO

[*IPC::SysV Class::Struct semget semctl semop*](#)

AUTHOR

Graham Barr <gbarr@pobox.com>

COPYRIGHT

Copyright (c) 1997 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

IPC::SysV – SysV IPC constants

SYNOPSIS

```
use IPC::SysV qw(IPC_STAT IPC_PRIVATE);
```

DESCRIPTION

IPC::SysV defines and conditionally exports all the constants defined in your system include files which are needed by the SysV IPC calls.

ftok(PATH, ID)

Return a key based on PATH and ID, which can be used as a key for msgget, semget and shmget.

See *ftok*

SEE ALSO

IPC::Msg, IPC::Semaphore, ftok

AUTHORS

Graham Barr <gbarr@pobox.com> Jarkko Hietaniemi <jhi@iki.fi>

COPYRIGHT

Copyright (c) 1997 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

less – perl pragma to request less of something from the compiler

SYNOPSIS

```
use less;  # unimplemented
```

DESCRIPTION

Currently unimplemented, this may someday be a compiler directive to make certain trade-offs, such as perhaps

```
use less 'memory';
use less 'CPU';
use less 'fat';
```

NAME

List::Util – A selection of general-utility list subroutines

SYNOPSIS

```
use List::Util qw(first max maxstr min minstr reduce shuffle sum);
```

DESCRIPTION

List::Util contains a selection of subroutines that people have expressed would be nice to have in the perl core, but the usage would not really be high enough to warrant the use of a keyword, and the size so small such that being individual extensions would be wasteful.

By default List::Util does not export any subroutines. The subroutines defined are

first BLOCK LIST

Similar to `grep` in that it evaluates `BLOCK` setting `$_` to each element of `LIST` in turn. `first` returns the first element where the result from `BLOCK` is a true value. If `BLOCK` never returns true or `LIST` was empty then `undef` is returned.

```
$foo = first { defined($_) } @list      # first defined value in @list
$foo = first { $_ > $value } @list      # first value in @list which
                                         # is greater than $value
```

This function could be implemented using `reduce` like this

```
$foo = reduce { defined($a) ? $a : wanted($b) ? $b : undef } undef, @list
```

for example `wanted()` could be `defined()` which would return the first defined value in `@list`

max LIST

Returns the entry in the list with the highest numerical value. If the list is empty then `undef` is returned.

```
$foo = max 1..10                # 10
$foo = max 3,9,12               # 12
$foo = max @bar, @baz           # whatever
```

This function could be implemented using `reduce` like this

```
$foo = reduce { $a > $b ? $a : $b } 1..10
```

maxstr LIST

Similar to `max`, but treats all the entries in the list as strings and returns the highest string as defined by the `gt` operator. If the list is empty then `undef` is returned.

```
$foo = maxstr 'A'..'Z'          # 'Z'
$foo = maxstr "hello","world"   # "world"
$foo = maxstr @bar, @baz        # whatever
```

This function could be implemented using `reduce` like this

```
$foo = reduce { $a gt $b ? $a : $b } 'A'..'Z'
```

min LIST

Similar to `max` but returns the entry in the list with the lowest numerical value. If the list is empty then `undef` is returned.

```
$foo = min 1..10                # 1
$foo = min 3,9,12               # 3
$foo = min @bar, @baz           # whatever
```

This function could be implemented using `reduce` like this

```
$foo = reduce { $a < $b ? $a : $b } 1..10
```

minstr LIST

Similar to min, but treats all the entries in the list as strings and returns the lowest string as defined by the lt operator. If the list is empty then undef is returned.

```
$foo = minstr 'A'..'Z'           # 'A'
$foo = minstr "hello","world"    # "hello"
$foo = minstr @bar, @baz         # whatever
```

This function could be implemented using reduce like this

```
$foo = reduce { $a lt $b ? $a : $b } 'A'..'Z'
```

reduce BLOCK LIST

Reduces LIST by calling BLOCK multiple times, setting \$a and \$b each time. The first call will be with \$a and \$b set to the first two elements of the list, subsequent calls will be done by setting \$a to the result of the previous call and \$b to the next element in the list.

Returns the result of the last call to BLOCK. If LIST is empty then undef is returned. If LIST only contains one element then that element is returned and BLOCK is not executed.

```
$foo = reduce { $a < $b ? $a : $b } 1..10      # min
$foo = reduce { $a lt $b ? $a : $b } 'aa'..'zz' # minstr
$foo = reduce { $a + $b } 1 .. 10              # sum
$foo = reduce { $a . $b } @bar                 # concat
```

shuffle LIST

Returns the elements of LIST in a random order

```
@cards = shuffle 0..51          # 0..51 in a random order
```

sum LIST

Returns the sum of all the elements in LIST.

```
$foo = sum 1..10                # 55
$foo = sum 3,9,12               # 24
$foo = sum @bar, @baz           # whatever
```

This function could be implemented using reduce like this

```
$foo = reduce { $a + $b } 1..10
```

KNOWN BUGS

With perl versions prior to 5.005 there are some cases where reduce will return an incorrect result. This will show up as test 7 of reduce.t failing.

SUGGESTED ADDITIONS

The following are additions that have been requested, but I have been reluctant to add due to them being very simple to implement in perl

```
# One argument is true
sub any { $_ && return 1 for @_; 0 }

# All arguments are true
sub all { $_ || return 0 for @_; 1 }

# All arguments are false
sub none { $_ && return 0 for @_; 1 }

# One argument is false
```

```
sub notall { $_ || return 1 for @_; 0 }  
# How many elements are true  
sub true { scalar grep { $_ } @_ }  
# How many elements are false  
sub false { scalar grep { !$_ } @_ }
```

COPYRIGHT

Copyright (c) 1997–2001 Graham Barr <gbarr@pobox.com>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

Scalar::Util – A selection of general-utility scalar subroutines

SYNOPSIS

```
use Scalar::Util qw(blessed dualvar isweak readonly reftype tainted weaken);
```

DESCRIPTION

Scalar::Util contains a selection of subroutines that people have expressed would be nice to have in the perl core, but the usage would not really be high enough to warrant the use of a keyword, and the size so small such that being individual extensions would be wasteful.

By default Scalar::Util does not export any subroutines. The subroutines defined are

blessed EXPR

If EXPR evaluates to a blessed reference the name of the package that it is blessed into is returned. Otherwise undef is returned.

```
$scalar = "foo";
$class  = blessed $scalar;           # undef

$ref    = [];
$class  = blessed $ref;              # undef

$obj    = bless [], "Foo";
$class  = blessed $obj;              # "Foo"
```

dualvar NUM, STRING

Returns a scalar that has the value NUM in a numeric context and the value STRING in a string context.

```
$foo = dualvar 10, "Hello";
$num = $foo + 2;           # 12
$str = $foo . " world";    # Hello world
```

isweak EXPR

If EXPR is a scalar which is a weak reference the result is true.

```
$ref = \ $foo;
$weak = isweak($ref);      # false
weaken($ref);
$weak = isweak($ref);      # true
```

openhandle FH

Returns FH if FH may be used as a filehandle and is open, or FH is a tied handle. Otherwise undef is returned.

```
$fh = openhandle(*STDIN);      # \*STDIN
$fh = openhandle(\*STDIN);    # \*STDIN
$fh = openhandle(*NOTOPEN);   # undef
$fh = openhandle("scalar");    # undef
```

readonly SCALAR

Returns true if SCALAR is readonly.

```
sub foo { readonly($_[0]) }

$readonly = foo($bar);        # false
$readonly = foo(0);           # true
```

reftype EXPR

If EXPR evaluates to a reference the type of the variable referenced is returned. Otherwise undef is returned.

```
$type = reftype "string";           # undef
$type = reftype \ $var;             # SCALAR
$type = reftype [];                # ARRAY

$obj = bless {}, "Foo";
$type = reftype $obj;               # HASH
```

tainted EXPR

Return true if the result of EXPR is tainted

```
$taint = tainted("constant");       # false
$taint = tainted($ENV{PWD});        # true if running under -T
```

weaken REF

REF will be turned into a weak reference. This means that it will not hold a reference count on the object it references. Also when the reference count on that object reaches zero, REF will be set to undef.

This is useful for keeping copies of references , but you don't want to prevent the object being DESTROY-ed at its usual time.

```
{
    my $var;
    $ref = \ $var;
    weaken($ref);                  # Make $ref a weak reference
}
# $ref is now undef
```

KNOWN BUGS

There is a bug in perl5.6.0 with UV's that are = 1<<31. This will show up as tests 8 and 9 of dualvar.t failing

COPYRIGHT

Copyright (c) 1997–2001 Graham Barr <gbarr@pobox.com>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Except weaken and isweak which are

Copyright (c) 1999 Tuomas J. Lukka <lukka@iki.fi>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as perl itself.

BLATANT PLUG

The weaken and isweak subroutines in this module and the patch to the core Perl were written in connection with the APress book 'Tuomas J. Lukka's Definitive Guide to Object-Oriented Programming in Perl', to avoid explaining why certain things would have to be done in cumbersome ways.

NAME

locale – Perl pragma to use and avoid POSIX locales for built-in operations

SYNOPSIS

```
@x = sort @y;      # ASCII sorting order
{
    use locale;
    @x = sort @y;   # Locale-defined sorting order
}
@x = sort @y;      # ASCII sorting order again
```

DESCRIPTION

This pragma tells the compiler to enable (or disable) the use of POSIX locales for built-in operations (LC_CTYPE for regular expressions, and LC_COLLATE for string comparison). Each "use locale" or "no locale" affects statements to the end of the enclosing BLOCK.

See [perllocale](#) for more detailed information on how Perl supports locales.

NAME

Locale::Constants – constants for Locale codes

SYNOPSIS

```
use Locale::Constants;  
  
$codeset = LOCALE_CODE_ALPHA_2;
```

DESCRIPTION

Locale::Constants defines symbols which are used in the four modules from the Locale–Codes distribution:

```
Locale::Language  
Locale::Country  
Locale::Currency  
Locale::Script
```

Note: at the moment only `Locale::Country` and `Locale::Script` support more than one code set.

The symbols defined are used to specify which codes you want to be used:

```
LOCALE_CODE_ALPHA_2  
LOCALE_CODE_ALPHA_3  
LOCALE_CODE_NUMERIC
```

You shouldn't have to use this module directly yourself – it is used by the three Locale modules, which in turn export the symbols.

KNOWN BUGS AND LIMITATIONS

None at the moment.

SEE ALSO

`Locale::Language`
Codes for identification of languages.

`Locale::Country`
Codes for identification of countries.

`Locale::Script`
Codes for identification of scripts.

`Locale::Currency`
Codes for identification of currencies and funds.

AUTHOR

Neil Bowers <neil@bowers.com>

COPYRIGHT

Copyright (C) 2002, Neil Bowers.

Copyright (C) 2001, Canon Research Centre Europe (CRE).

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

Locale::Country – ISO codes for country identification (ISO 3166)

SYNOPSIS

```
use Locale::Country;

$country = code2country('jp');      # $country gets 'Japan'
$code    = country2code('Norway');  # $code gets 'no'

@codes    = all_country_codes();
@names    = all_country_names();

# semi-private routines
Locale::Country::alias_code('uk' => 'gb');
Locale::Country::rename_country('gb' => 'Great Britain');
```

DESCRIPTION

The `Locale::Country` module provides access to the ISO codes for identifying countries, as defined in ISO 3166-1. You can either access the codes via the [conversion routines](#) (described below), or with the two functions which return lists of all country codes or all country names.

There are three different code sets you can use for identifying countries:

alpha-2

Two letter codes, such as 'tv' for Tuvalu. This code set is identified with the symbol `LOCALE_CODE_ALPHA_2`.

alpha-3

Three letter codes, such as 'brb' for Barbados. This code set is identified with the symbol `LOCALE_CODE_ALPHA_3`.

numeric

Numeric codes, such as 064 for Bhutan. This code set is identified with the symbol `LOCALE_CODE_NUMERIC`.

All of the routines take an optional additional argument which specifies the code set to use. If not specified, it defaults to the two-letter codes. This is partly for backwards compatibility (previous versions of this module only supported the alpha-2 codes), and partly because they are the most widely used codes.

The alpha-2 and alpha-3 codes are not case-dependent, so you can use 'BO', 'Bo', 'bo' or 'bo' for Bolivia. When a code is returned by one of the functions in this module, it will always be lower-case.

As of version 2.00, `Locale::Country` supports variant names for countries. So, for example, the country code for "United States" is "us", so `country2code('United States')` returns 'us'. Now the following will also return 'us':

```
country2code('United States of America')
country2code('USA')
```

CONVERSION ROUTINES

There are three conversion routines: `code2country()`, `country2code()`, and `country_code2code()`.

`code2country(CODE, [CODESET])`

This function takes a country code and returns a string which contains the name of the country identified. If the code is not a valid country code, as defined by ISO 3166, then `undef` will be returned:

```
$country = code2country('fi');
```

`country2code(STRING, [CODESET])`

This function takes a country name and returns the corresponding country code, if such exists. If the argument could not be identified as a country name, then `undef` will be returned:

```
$code = country2code('Norway', LOCALE_CODE_ALPHA_3);
# $code will now be 'nor'
```

The case of the country name is not important. See the section [KNOWN BUGS AND LIMITATIONS](#) below.

`country_code2code(CODE, CODESET, CODESET)`

This function takes a country code from one code set, and returns the corresponding code from another code set.

```
$alpha2 = country_code2code('fin',
                             LOCALE_CODE_ALPHA_3, LOCALE_CODE_ALPHA_2);
# $alpha2 will now be 'fi'
```

If the code passed is not a valid country code in the first code set, or if there isn't a code for the corresponding country in the second code set, then `undef` will be returned.

QUERY ROUTINES

There are two function which can be used to obtain a list of all codes, or all country names:

`all_country_codes([CODESET])`

Returns a list of all two-letter country codes. The codes are guaranteed to be all lower-case, and not in any particular order.

`all_country_names([CODESET])`

Returns a list of all country names for which there is a corresponding country code in the specified code set. The names are capitalised, and not returned in any particular order.

Not all countries have alpha-3 and numeric codes – some just have an alpha-2 code, so you'll get a different number of countries depending on which code set you specify.

SEMI-PRIVATE ROUTINES

`Locale::Country` provides two semi-private routines for modifying the internal data. Given their status, they aren't exported by default, and so need to be called by prefixing the function name with the package name.

`alias_code`

Define a new code as an alias for an existing code:

```
Locale::Country::alias_code( ALIAS => CODE [, CODESET ] )
```

This feature was added as a mechanism for handling a "uk" code. The ISO standard says that the two-letter code for "United Kingdom" is "gb", whereas domain names are all .uk.

By default the module does not understand "uk", since it is implementing an ISO standard. If you would like 'uk' to work as the two-letter code for United Kingdom, use the following:

```
Locale::Country::alias_code('uk' => 'gb');
```

With this code, both "uk" and "gb" are valid codes for United Kingdom, with the reverse lookup returning "uk" rather than the usual "gb".

Note: this function was previously called `_alias_code`, but the leading underscore has been dropped. The old name will be supported for all 2.X releases for backwards compatibility.

`rename_country`

If the official country name just isn't good enough for you, you can rename a country. For example, the official country name for code 'gb' is 'United Kingdom'. If you want to change that, you might call:

```
Locale::Country::rename_country('gb' => 'Great Britain');
```

This means that calling `code2country('gb')` will now return 'Great Britain' instead of 'United Kingdom'. The original country name is retained as an alias, so for the above example, `country2code('United Kingdom')` will still return 'gb'.

EXAMPLES

The following example illustrates use of the `code2country()` function. The user is prompted for a country code, and then told the corresponding country name:

```
$| = 1;    # turn off buffering

print "Enter country code: ";
chop($code = <STDIN>);
$country = code2country($code, LOCALE_CODE_ALPHA_2);
if (defined $country)
{
    print "$code = $country\n";
}
else
{
    print "'$code' is not a valid country code!\n";
}
```

DOMAIN NAMES

Most top-level domain names are based on these codes, but there are certain codes which aren't. If you are using this module to identify country from hostname, your best bet is to preprocess the country code.

For example, **edu**, **com**, **gov** and friends would map to **us**; **uk** would map to **gb**. Any others?

KNOWN BUGS AND LIMITATIONS

- When using `country2code()`, the country name must currently appear exactly as it does in the source of the module. The module now supports a small number of variants.
Possible extensions to this are: an interface for getting at the list of variant names, and regular expression matches.
- In the current implementation, all data is read in when the module is loaded, and then held in memory. A lazy implementation would be more memory friendly.
- Support for country names in different languages.

SEE ALSO

`Locale::Language`

ISO two letter codes for identification of language (ISO 639).

`Locale::Script`

ISO codes for identification of scripts (ISO 15924).

`Locale::Currency`

ISO three letter codes for identification of currencies and funds (ISO 4217).

`Locale::SubCountry`

ISO codes for country sub-divisions (states, counties, provinces, etc), as defined in ISO 3166-2. This module is not part of the `Locale-Codes` distribution, but is available from CPAN in `CPAN/modules/by-module/Locale/`

ISO 3166-1

The ISO standard which defines these codes.

<http://www.iso.org/iso/en/prods-services/iso3166ma/index.html>

Official home page for the ISO 3166 maintenance agency.

<http://www.egt.ie/standards/iso3166/iso3166-1-en.html>

Another useful, but not official, home page.

<http://www.cia.gov/cia/publications/factbook/docs/app-d-1.html>

An appendix in the CIA world fact book which lists country codes as defined by ISO 3166, FIPS 10-4, and internet domain names.

AUTHOR

Neil Bowers <neil@bowers.com>

COPYRIGHT

Copyright (C) 2002, Neil Bowers.

Copyright (c) 1997-2001 Canon Research Centre Europe (CRE).

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

Locale::Currency – ISO three letter codes for currency identification (ISO 4217)

SYNOPSIS

```
use Locale::Currency;

$curr = code2currency('usd');    # $curr gets 'US Dollar'
$code = currency2code('Euro');   # $code gets 'eur'

@codes = all_currency_codes();
@names = all_currency_names();
```

DESCRIPTION

The `Locale::Currency` module provides access to the ISO three-letter codes for identifying currencies and funds, as defined in ISO 4217. You can either access the codes via the [conversion routines](#) (described below), or with the two functions which return lists of all currency codes or all currency names.

There are two special codes defined by the standard which aren't understood by this module:

XTS

Specifically reserved for testing purposes.

XXX

For transactions where no currency is involved.

CONVERSION ROUTINES

There are two conversion routines: `code2currency()` and `currency2code()`.

`code2currency()`

This function takes a three letter currency code and returns a string which contains the name of the currency identified. If the code is not a valid currency code, as defined by ISO 4217, then `undef` will be returned.

```
$curr = code2currency($code);
```

`currency2code()`

This function takes a currency name and returns the corresponding three letter currency code, if such exists. If the argument could not be identified as a currency name, then `undef` will be returned.

```
$code = currency2code('French Franc');
```

The case of the currency name is not important. See the section [KNOWN BUGS AND LIMITATIONS](#) below.

QUERY ROUTINES

There are two function which can be used to obtain a list of all currency codes, or all currency names:

`all_currency_codes()`

Returns a list of all three-letter currency codes. The codes are guaranteed to be all lower-case, and not in any particular order.

`all_currency_names()`

Returns a list of all currency names for which there is a corresponding three-letter currency code. The names are capitalised, and not returned in any particular order.

EXAMPLES

The following example illustrates use of the `code2currency()` function. The user is prompted for a currency code, and then told the corresponding currency name:

```
$| = 1;    # turn off buffering
```

```
print "Enter currency code: ";
chop($code = <STDIN>);
$curr = code2currency($code);
if (defined $curr)
{
    print "$code = $curr\n";
}
else
{
    print "'$code' is not a valid currency code!\n";
}
```

KNOWN BUGS AND LIMITATIONS

- In the current implementation, all data is read in when the module is loaded, and then held in memory. A lazy implementation would be more memory friendly.
- This module also includes the special codes which are not for a currency, such as Gold, Platinum, etc. This might cause a problem if you're using this module to display a list of currencies. Let Neil know if this does cause a problem, and we can do something about it.
- ISO 4217 also defines a numeric code for each currency. Currency codes are not currently supported by this module, in the same way Locale::Country supports multiple codesets.
- There are three cases where there is more than one code for the same currency name. Kwacha has two codes: mwk for Malawi, and zmk for Zambia. The Russian Ruble has two codes: rub and rur. The Belarussian Ruble has two codes: byr and byb. The `currency2code()` function only returns one code, so you might not get back the code you expected.

SEE ALSO

Locale::Country

ISO codes for identification of country (ISO 3166).

Locale::Script

ISO codes for identification of written scripts (ISO 15924).

ISO 4217:1995

Code for the representation of currencies and funds.

<http://www.bsi-global.com/iso4217currency>

Official web page for the ISO 4217 maintenance agency. This has the latest list of codes, in MS Word format. Boo.

AUTHOR

Michael Hennecke <hennecke@rz.uni-karlsruhe.de> and Neil Bowers <neil@bowers.com>

COPYRIGHT

Copyright (C) 2002, Neil Bowers.

Copyright (c) 2001 Michael Hennecke and Canon Research Centre Europe (CRE).

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

Locale::Language – ISO two letter codes for language identification (ISO 639)

SYNOPSIS

```
use Locale::Language;

$lang = code2language('en');      # $lang gets 'English'
$code = language2code('French');  # $code gets 'fr'

@codes = all_language_codes();
@names = all_language_names();
```

DESCRIPTION

The `Locale::Language` module provides access to the ISO two-letter codes for identifying languages, as defined in ISO 639. You can either access the codes via the *conversion routines* (described below), or via the two functions which return lists of all language codes or all language names.

CONVERSION ROUTINES

There are two conversion routines: `code2language()` and `language2code()`.

`code2language()`

This function takes a two letter language code and returns a string which contains the name of the language identified. If the code is not a valid language code, as defined by ISO 639, then `undef` will be returned.

```
$lang = code2language($code);
```

`language2code()`

This function takes a language name and returns the corresponding two letter language code, if such exists. If the argument could not be identified as a language name, then `undef` will be returned.

```
$code = language2code('French');
```

The case of the language name is not important. See the section *KNOWN BUGS AND LIMITATIONS* below.

QUERY ROUTINES

There are two function which can be used to obtain a list of all language codes, or all language names:

`all_language_codes()`

Returns a list of all two-letter language codes. The codes are guaranteed to be all lower-case, and not in any particular order.

`all_language_names()`

Returns a list of all language names for which there is a corresponding two-letter language code. The names are capitalised, and not returned in any particular order.

EXAMPLES

The following example illustrates use of the `code2language()` function. The user is prompted for a language code, and then told the corresponding language name:

```
$| = 1;      # turn off buffering

print "Enter language code: ";
chop($code = <STDIN>);
$lang = code2language($code);
if (defined $lang)
{
    print "$code = $lang\n";
}
```



```
else
{
    print "'$code' is not a valid language code!\n";
}
```

KNOWN BUGS AND LIMITATIONS

- In the current implementation, all data is read in when the module is loaded, and then held in memory. A lazy implementation would be more memory friendly.
- Currently just supports the two letter language codes – there are also three-letter codes, and numbers. Would these be of any use to anyone?

SEE ALSO

Locale::Country

ISO codes for identification of country (ISO 3166). Supports 2-letter, 3-letter, and numeric country codes.

Locale::Script

ISO codes for identification of written scripts (ISO 15924).

Locale::Currency

ISO three letter codes for identification of currencies and funds (ISO 4217).

ISO 639:1988 (E/F)

Code for the representation of names of languages.

<http://lcweb.loc.gov/standards/iso639-2/langhome.html>

Home page for ISO 639-2.

AUTHOR

Neil Bowers <neil@bowers.com>

COPYRIGHT

Copyright (C) 2002, Neil Bowers.

Copyright (c) 1997–2001 Canon Research Centre Europe (CRE).

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

Locale::Maketext — framework for localization

SYNOPSIS

```
package MyProgram;
use strict;
use MyProgram::L10N;
# ...which inherits from Locale::Maketext
my $lh = MyProgram::L10N->get_handle() || die "What language?";
...
# And then any messages your program emits, like:
warn $lh->maketext( "Can't open file [_1]: [_2]\n", $f, $! );
...
```

DESCRIPTION

It is a common feature of applications (whether run directly, or via the Web) for them to be "localized" — i.e., for them to present an English interface to an English-speaker, a German interface to a German-speaker, and so on for all languages it's programmed with. `Locale::Maketext` is a framework for software localization; it provides you with the tools for organizing and accessing the bits of text and text-processing code that you need for producing localized applications.

In order to make sense of `Maketext` and how all its components fit together, you should probably go read [Locale::Maketext::TPJ13](#)/[Locale::Maketext::TPJ13](#), and then read the following documentation.

You may also want to read over the source for `File::Findgrep` and its constituent modules — they are a complete (if small) example application that uses `Maketext`.

QUICK OVERVIEW

The basic design of `Locale::Maketext` is object-oriented, and `Locale::Maketext` is an abstract base class, from which you derive a "project class". The project class (with a name like `"TkBocciBall::Localize"`, which you then use in your module) is in turn the base class for all the "language classes" for your project (with names `"TkBocciBall::Localize::it"`, `"TkBocciBall::Localize::en"`, `"TkBocciBall::Localize::fr"`, etc.).

A language class is a class containing a lexicon of phrases as class data, and possibly also some methods that are of use in interpreting phrases in the lexicon, or otherwise dealing with text in that language.

An object belonging to a language class is called a "language handle"; it's typically a flyweight object.

The normal course of action is to call:

```
use TkBocciBall::Localize; # the localization project class
$lh = TkBocciBall::Localize->get_handle();
# Depending on the user's locale, etc., this will
# make a language handle from among the classes available,
# and any defaults that you declare.
die "Couldn't make a language handle?" unless $lh;
```

From then on, you use the `maketext` function to access entries in whatever lexicon(s) belong to the language handle you got. So, this:

```
print $lh->maketext("You won!"), "\n";
```

...emits the right text for this language. If the object in `$lh` belongs to class `"TkBocciBall::Localize::fr"` and `%TkBocciBall::Localize::fr::Lexicon` contains `("You won!" => "Tu as gagné!")`, then the above code happily tells the user "Tu as gagné!".

METHODS

`Locale::Maketext` offers a variety of methods, which fall into three categories:

- Methods to do with constructing language handles.
- maketext and other methods to do with accessing %Lexicon data for a given language handle.
- Methods that you may find it handy to use, from routines of yours that you put in %Lexicon entries.

These are covered in the following section.

Construction Methods

These are to do with constructing a language handle:

- `$lh = YourProjClass->get_handle(...langtags...) || die "lg-handle?"`;

This tries loading classes based on the language-tags you give (like ("en-US", "sk", "kon", "es-MX", "ja", "i-klingon")), and for the first class that succeeds, returns `YourProjClass::language-new()`.

It runs thru the entire given list of language-tags, and finds no classes for those exact terms, it then tries "superordinate" language classes. So if no "en-US" class (i.e., `YourProjClass::en_us`) was found, nor classes for anything else in that list, we then try its superordinate, "en" (i.e., `YourProjClass::en`), and so on thru the other language-tags in the given list: "es". (The other language-tags in our example list: happen to have no superordinates.)

If none of those language-tags leads to loadable classes, we then try classes derived from `YourProjClass->fallback_languages()` and then if nothing comes of that, we use classes named by `YourProjClass->fallback_language_classes()`. Then in the (probably quite unlikely) event that that fails, we just return undef.

- `$lh = YourProjClass->get_handle() || die "lg-handle?"`;

When `get_handle` is called with an empty parameter list, magic happens:

If `get_handle` senses that it's running in program that was invoked as a CGI, then it tries to get language-tags out of the environment variable "HTTP_ACCEPT_LANGUAGE", and it pretends that those were the languages passed as parameters to `get_handle`.

Otherwise (i.e., if not a CGI), this tries various OS-specific ways to get the language-tags for the current locale/language, and then pretends that those were the value(s) passed to `get_handle`.

Currently this OS-specific stuff consists of looking in the environment variables "LANG" and "LANGUAGE"; and on MSWin machines (where those variables are typically unused), this also tries using the module `Win32::Locale` to get a language-tag for whatever language/locale is currently selected in the "Regional Settings" (or "International"?) Control Panel. I welcome further suggestions for making this do the Right Thing under other operating systems that support localization.

If you're using localization in an application that keeps a configuration file, you might consider something like this in your project class:

```
sub get_handle_via_config {
    my $class = $_[0];
    my $preferred_language = $Config_settings{'language'};
    my $lh;
    if($preferred_language) {
        $lh = $class->get_handle($chosen_language)
        || die "No language handle for \"$chosen_language\" or the like";
    } else {
        # Config file missing, maybe?
        $lh = $class->get_handle()
        || die "Can't get a language handle";
    }
    return $lh;
}
```

```
}
```

- `$lh = YourProjClass::langname-new();`

This constructs a language handle. You usually **don't** call this directly, but instead let `get_handle` find a language class to use and to then call `-new` on.

- `$lh-init();`

This is called by `-new` to initialize newly-constructed language handles. If you define an `init` method in your class, remember that it's usually considered a good idea to call `$lh-SUPER::init` in it (presumably at the beginning), so that all classes get a chance to initialize a new object however they see fit.

- `YourProjClass-fallback_languages()`

`get_handle` appends the return value of this to the end of whatever list of languages you pass `get_handle`. Unless you override this method, your project class will inherit `Locale::Maketext's fallback_languages`, which currently returns `('i-default', 'en', 'en-US')`. ("i-default" is defined in RFC 2277).

This method (by having it return the name of a language-tag that has an existing language class) can be used for making sure that `get_handle` will always manage to construct a language handle (assuming your language classes are in an appropriate `@INC` directory). Or you can use the next method:

- `YourProjClass-fallback_language_classes()`

`get_handle` appends the return value of this to the end of the list of classes it will try using. Unless you override this method, your project class will inherit `Locale::Maketext's fallback_language_classes`, which currently returns an empty list, `()`. By setting this to some value (namely, the name of a loadable language class), you can be sure that `get_handle` will always manage to construct a language handle.

The "maketext" Method

This is the most important method in `Locale::Maketext`:

```
$text = $lh-maketext(key, ...parameters for this phrase...);
```

This looks in the `%Lexicon` of the language handle `$lh` and all its superclasses, looking for an entry whose key is the string `key`. Assuming such an entry is found, various things then happen, depending on the value found:

If the value is a scalarref, the scalar is dereferenced and returned (and any parameters are ignored). If the value is a coderef, we return `&$value($lh, ...parameters...)`. If the value is a string that *doesn't* look like it's in Bracket Notation, we return it (after replacing it with a scalarref, in its `%Lexicon`). If the value *does* look like it's in Bracket Notation, then we compile it into a sub, replace the string in the `%Lexicon` with the new coderef, and then we return `&$new_sub($lh, ...parameters...)`.

Bracket Notation is discussed in a later section. Note that trying to compile a string into Bracket Notation can throw an exception if the string is not syntactically valid (say, by not balancing brackets right.)

Also, calling `&$coderef($lh, ...parameters...)` can throw any sort of exception (if, say, code in that sub tries to divide by zero). But a very common exception occurs when you have Bracket Notation text that says to call a method "foo", but there is no such method. (E.g., "You have [quatn,1,ball]." will throw an exception on trying to call `$lh-quatn($_[1], 'ball')` — you presumably meant "quant".) `maketext` catches these exceptions, but only to make the error message more readable, at which point it rethrows the exception.

An exception *may* be thrown if `key` is not found in any of `$lh's %Lexicon` hashes. What happens if a key is not found, is discussed in a later section, "Controlling Lookup Failure".

Note that you might find it useful in some cases to override the `maketext` method with an "after method", if you want to translate encodings, or even scripts:

```
package YrProj::zh_cn; # Chinese with PRC-style glyphs
use base ('YrProj::zh_tw'); # Taiwan-style
sub maketext {
    my $self = shift(@_);
    my $value = $self->maketext(@_);
    return Chineeze::taiwan2mainland($value);
}
```

Or you may want to override it with something that traps any exceptions, if that's critical to your program:

```
sub maketext {
    my($lh, @stuff) = @_;
    my $out;
    eval { $out = $lh->SUPER::maketext(@stuff) };
    return $out unless $@;
    ...otherwise deal with the exception...
}
```

Other than those two situations, I don't imagine that it's useful to override the `maketext` method. (If you run into a situation where it is useful, I'd be interested in hearing about it.)

`$lh-fail_with` or `$lh-fail_with(PARAM)`
`$lh-failure_handler_auto`

These two methods are discussed in the section "Controlling Lookup Failure".

Utility Methods

These are methods that you may find it handy to use, generally from %Lexicon routines of yours (whether expressed as Bracket Notation or not).

```
$language-quant($number, $singular)
$language-quant($number, $singular, $plural)
$language-quant($number, $singular, $plural, $negative)
```

This is generally meant to be called from inside Bracket Notation (which is discussed later), as in

```
"Your search matched [quant,_1,document]!"
```

It's for *quantifying* a noun (i.e., saying how much of it there is, while giving the current form of it). The behavior of this method is handy for English and a few other Western European languages, and you should override it for languages where it's not suitable. You can feel free to read the source, but the current implementation is basically as this pseudocode describes:

```
if $number is 0 and there's a $negative,
    return $negative;
elsif $number is 1,
    return "1 $singular";
elsif there's a $plural,
    return "$number $plural";
else
    return "$number " . $singular . "s";
#
# ...except that we actually call numf to
# stringify $number before returning it.
```

So for English (with Bracket Notation) "...[quant,_1,file]..." is fine (for 0 it returns "0 files", for 1 it returns "1 file", and for more it returns "2 files", etc.)

But for "directory", you'd want "[quant,_1,direcory,directories]" so that our elementary quant method doesn't think that the plural of "directory" is "directorys". And you might find that the output may sound better if you specify a negative form, as in:

```
"[quant,_1,file,files,No files] matched your query.\n"
```

Remember to keep in mind verb agreement (or adjectives too, in other languages), as in:

```
"[quant,_1,document] were matched.\n"
```

Because if _1 is one, you get "1 document **were** matched". An acceptable hack here is to do something like this:

```
"[quant,_1,document was, documents were] matched.\n"
```

`$language-numf($number)`

This returns the given number formatted nicely according to this language's conventions. Maketext's default method is mostly to just take the normal string form of the number (applying `sprintf "%G"` for only very large numbers), and then to add commas as necessary. (Except that we apply `tr/,./././` if `$language-{numf_comma}` is true; that's a bit of a hack that's useful for languages that express two million as "2.000.000" and not as "2,000,000").

If you want anything fancier, consider overriding this with something that uses [Number::Format/Number::Format](#), or does something else entirely.

Note that `numf` is called by `quant` for stringifying all quantifying numbers.

`$language-sprintf($format, @items)`

This is just a wrapper around Perl's normal `sprintf` function. It's provided so that you can use "sprintf" in Bracket Notation:

```
"Couldn't access datanode [sprintf,%10x=~[%s~],_1,_2]!\n"
```

returning...

```
Couldn't access datanode          Stuff=[thangamabob]!
```

`$language-language_tag()`

Currently this just takes the last bit of `ref($language)`, turns underscores to dashes, and returns it. So if `$language` is an object of class `Hee::HOO::Haw::en_us`, `$language-language_tag()` returns "en-us". (Yes, the usual representation for that language tag is "en-US", but case is *never* considered meaningful in language-tag comparison.)

You may override this as you like; Maketext doesn't use it for anything.

`$language-encoding()`

Currently this isn't used for anything, but it's provided (with default value of `(ref($language) && $language->{'encoding'})` or "iso-8859-1") as a sort of suggestion that it may be useful/necessary to associate encodings with your language handles (whether on a per-class or even per-handle basis.)

Language Handle Attributes and Internals

A language handle is a flyweight object — i.e., it doesn't (necessarily) carry any data of interest, other than just being a member of whatever class it belongs to.

A language handle is implemented as a blessed hash. Subclasses of yours can store whatever data you want in the hash. Currently the only hash entry used by any crucial Maketext method is "fail", so feel free to use anything else as you like.

Remember: Don't be afraid to read the Maketext source if there's any point on which this documentation is unclear. This documentation is vastly longer than the module source itself.

LANGUAGE CLASS HIERARCHIES

These are Locale::Maketext's assumptions about the class hierarchy formed by all your language classes:

- You must have a project base class, which you load, and which you then use as the first argument in the call to `YourProjClass->get_handle(...)`. It should derive (whether directly or indirectly) from `Locale::Maketext`. It **doesn't matter** how you name this class, altho assuming this is the localization component of your Super Mega Program, good names for your project class might be `SuperMegaProgram::Localization`, `SuperMegaProgram::L10N`, `SuperMegaProgram::I18N`, `SuperMegaProgram::International`, or even `SuperMegaProgram::Languages` or `SuperMegaProgram::Messages`.
- Language classes are what `YourProjClass->get_handle` will try to load. It will look for them by taking each language-tag (**skipping** it if it doesn't look like a language-tag or locale-tag!), turning it to all lowercase, turning and dashes to underscores, and appending it to `YourProjClass . "::"`. So this:

```
$lh = YourProjClass->get_handle(
    'en-US', 'fr', 'kon', 'i-klingon', 'i-klingon-romanized'
);
```

will try loading the classes `YourProjClass::en_us` (note lowercase!), `YourProjClass::fr`, `YourProjClass::kon`, `YourProjClass::i_klingon` and `YourProjClass::i_klingon_romanized`. (And it'll stop at the first one that actually loads.)

- I assume that each language class derives (directly or indirectly) from your project class, and also defines its `@ISA`, its `%Lexicon`, or both. But I anticipate no dire consequences if these assumptions do not hold.
- Language classes may derive from other language classes (altho they should have "use *Thatclassname*" or "use base qw(...*classes*...)"). They may derive from the project class. They may derive from some other class altogether. Or via multiple inheritance, it may derive from any mixture of these.
- I foresee no problems with having multiple inheritance in your hierarchy of language classes. (As usual, however, Perl will complain bitterly if you have a cycle in the hierarchy: i.e., if any class is its own ancestor.)

ENTRIES IN EACH LEXICON

A typical `%Lexicon` entry is meant to signify a phrase, taking some number (0 or more) of parameters. An entry is meant to be accessed by via a string *key* in `$lh->maketext(key, ...parameters...)`, which should return a string that is generally meant for be used for "output" to the user — regardless of whether this actually means printing to STDOUT, writing to a file, or putting into a GUI widget.

While the key must be a string value (since that's a basic restriction that Perl places on hash keys), the value in the lexicon can currently be of several types: a defined scalar, `scalarref`, or `coderef`. The use of these is explained above, in the section "The "maketext" Method", and Bracket Notation for strings is discussed in the next section.

While you can use arbitrary unique IDs for lexicon keys (like `"_min_larger_max_error"`), it is often useful for if an entry's key is itself a valid value, like this example error message:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
```

Compare this code that uses an arbitrary ID...

```
die $lh->maketext( "_min_larger_max_error", $min, $max )
    if $min > $max;
```

...to this code that uses a key-as-value:

```
die $lh->maketext(
    "Minimum ([_1]) is larger than maximum ([_2])!\n",
    $min, $max
```

```
) if $min > $max;
```

The second is, in short, more readable. In particular, it's obvious that the number of parameters you're feeding to that phrase (two) is the number of parameters that it *wants* to be fed. (Since you see `_1` and a `_2` being used in the key there.)

Also, once a project is otherwise complete and you start to localize it, you can scrape together all the various keys you use, and pass it to a translator; and then the translator's work will go faster if what he's presented is this:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
=> "",    # fill in something here, Jacques!
```

rather than this more cryptic mess:

```
"_min_larger_max_error"
=> "",    # fill in something here, Jacques
```

I think that keys as lexicon values makes the completed lexicon entries more readable:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
=> "Le minimum ([_1]) est plus grand que le maximum ([_2])!\n",
```

Also, having valid values as keys becomes very useful if you set up an `_AUTO` lexicon. `_AUTO` lexicons are discussed in a later section.

I almost always use keys that are themselves valid lexicon values. One notable exception is when the value is quite long. For example, to get the screenful of data that a command-line program might return when given an unknown switch, I often just use a key `"_USAGE_MESSAGE"`. At that point I then go and immediately to define that lexicon entry in the `ProjectClass::L10N::en` lexicon (since English is always my "project language"):

```
'_USAGE_MESSAGE' => <<'EOSTUFF',
...long long message...
EOSTUFF
```

and then I can use it as:

```
getopt('oDI', \%opts) or die $lh->maketext('_USAGE_MESSAGE');
```

Incidentally, note that each class's `%Lexicon` inherits—and—extends the lexicons in its superclasses. This is not because these are special hashes *per se*, but because you access them via the `maketext` method, which looks for entries across all the `%Lexicon`'s in a language class *and* all its ancestor classes. (This is because the idea of "class data" isn't directly implemented in Perl, but is instead left to individual class-systems to implement as they see fit..)

Note that you may have things stored in a lexicon besides just phrases for output: for example, if your program takes input from the keyboard, asking a "(Y/N)" question, you probably need to know what equivalent of "Y[es]/N[o]" is in whatever language. You probably also need to know what the equivalents of the answers "y" and "n" are. You can store that information in the lexicon (say, under the keys `"~answer_y"` and `"~answer_n"`, and the long forms as `"~answer_yes"` and `"~answer_no"`, where `"~"` is just an ad-hoc character meant to indicate to programmers/translators that these are not phrases for output).

Or instead of storing this in the language class's lexicon, you can (and, in some cases, really should) represent the same bit of knowledge as code is a method in the language class. (That leaves a tidy distinction between the lexicon as the things we know how to *say*, and the rest of the things in the lexicon class as things that we know how to *do*.) Consider this example of a processor for responses to French "oui/non" questions:

```
sub y_or_n {
    return undef unless defined $_[1] and length $_[1];
    my $answer = lc $_[1]; # smash case
```



```

    return 1 if $answer eq 'o' or $answer eq 'oui';
    return 0 if $answer eq 'n' or $answer eq 'non';
    return undef;
}

```

...which you'd then call in a construct like this:

```

my $response;
until(defined $response) {
    print $lh->maketext("Open the pod bay door (y/n)? ");
    $response = $lh->y_or_n( get_input_from_keyboard_somewhat() );
}
if($response) { $pod_bay_door->open() }
else          { $pod_bay_door->leave_closed() }

```

Other data worth storing in a lexicon might be things like filenames for language-targeted resources:

```

...
"_main_splash_png"
=> "/styles/en_us/main_splash.png",
"_main_splash_imagemap"
=> "/styles/en_us/main_splash.incl",
"_general_graphics_path"
=> "/styles/en_us/",
"_alert_sound"
=> "/styles/en_us/hey_there.wav",
"_forward_icon"
=> "left_arrow.png",
"_backward_icon"
=> "right_arrow.png",
# In some other languages, left equals
# BACKwards, and right is FOREwards.
...

```

You might want to do the same thing for expressing key bindings or the like (since hardwiring "q" as the binding for the function that quits a screen/menu/program is useful only if your language happens to associate "q" with "quit"!)

BRACKET NOTATION

Bracket Notation is a crucial feature of `Locale::Maketext`. I mean Bracket Notation to provide a replacement for `sprintf` formatting. Everything you do with Bracket Notation could be done with a sub block, but bracket notation is meant to be much more concise.

Bracket Notation is like a miniature "template" system (in the sense of [Text::Template/Text::Template](#), not in the sense of C++ templates), where normal text is passed thru basically as is, but text in special regions is specially interpreted. In Bracket Notation, you use brackets ("[...] — not "{...}"!) to note sections that are specially interpreted.

For example, here all the areas that are taken literally are underlined with a "^", and all the in-bracket special regions are underlined with an X:

```

"Minimum ([_1]) is larger than maximum ([_2])!\n",
^^^^^^^^^ XX ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ XX ^^^^^

```

When that string is compiled from bracket notation into a real Perl sub, it's basically turned into:

```

sub {
    my $lh = $_[0];
    my @params = @_;
    return join ' ',

```

```

    "Minimum (",
    ...some code here...
    ") is larger than maximum (",
    ...some code here...
    ")!\n",
}
# to be called by $lh->maketext(KEY, params...)

```

In other words, text outside bracket groups is turned into string literals. Text in brackets is rather more complex, and currently follows these rules:

- Bracket groups that are empty, or which consist only of whitespace, are ignored. (Examples: "[]", "[]", or a [and a] with returns and/or tabs and/or spaces between them.
Otherwise, each group is taken to be a comma-separated group of items, and each item is interpreted as follows:
- An item that is "*_digits*" or "*_*-*digits*" is interpreted as `$_[value]`. I.e., "`_1`" becomes with `$_[1]`, and "`_-3`" is interpreted as `$_[-3]` (in which case `@_` should have at least three elements in it). Note that `$_[0]` is the language handle, and is typically not named directly.
- An item "`_*`" is interpreted to mean "all of `@_` except `$_[0]`". I.e., `@_[1..$#_]`. Note that this is an empty list in the case of calls like `$lh->maketext(key)` where there are no parameters (except `$_[0]`, the language handle).
- Otherwise, each item is interpreted as a string literal.

The group as a whole is interpreted as follows:

- If the first item in a bracket group looks like a method name, then that group is interpreted like this:

```

    $lh->that_method_name(
        ...rest of items in this group...
    ),

```
- If the first item in a bracket group is "`*`", it's taken as shorthand for the so commonly called "quant" method. Similarly, if the first item in a bracket group is "`#`", it's taken to be shorthand for "numf".
- If the first item in a bracket group is empty-string, or "`_*`" or "*_digits*" or "*_*-*digits*", then that group is interpreted as just the interpolation of all its items:

```

    join('',
        ...rest of items in this group...
    ),

```

Examples: "`[_1]`" and "`[,_1]`", which are synonymous; and "`[ID-(,_4,-,_2,)]`", which compiles as `join "", "ID-(", $_[4], "-", $_[2], ")"`.
- Otherwise this bracket group is invalid. For example, in the group "`[!@#,whatever]`", the first item "`!@#`" is neither empty-string, "*_number*", "*_*-*number*", "`_*`", nor a valid method name; and so Locale::Maketext will throw an exception of you try compiling an expression containing this bracket group.

Note, incidentally, that items in each group are comma-separated, not `/\s*, \s*/`-separated. That is, you might expect that this bracket group:

```
"Hoohah [foo, _1 , bar ,baz]!"
```

would compile to this:

```

sub {
    my $lh = $_[0];
    return join ' ',

```

```

    "Hoohah ",
    $lh->foo( $_[1], "bar", "baz"),
    "!",
}

```

But it actually compiles as this:

```

sub {
    my $lh = $_[0];
    return join '',
        "Hoohah ",
        $lh->foo(" _1 ", " bar ", "baz"),    #!!!
        "!",
}

```

In the notation discussed so far, the characters "[" and "]" are given special meaning, for opening and closing bracket groups, and "," has a special meaning inside bracket groups, where it separates items in the group. This begs the question of how you'd express a literal "[" or "]" in a Bracket Notation string, and how you'd express a literal comma inside a bracket group. For this purpose I've adopted "~" (tilde) as an escape character: "~[" means a literal '[' character anywhere in Bracket Notation (i.e., regardless of whether you're in a bracket group or not), and ditto for "~]" meaning a literal ']', and "~," meaning a literal comma. (Altho "~," means a literal comma outside of bracket groups — it's only inside bracket groups that commas are special.)

And on the off chance you need a literal tilde in a bracket expression, you get it with "~~".

Currently, an unescaped "~" before a character other than a bracket or a comma is taken to mean just a "~" and that character. I.e., "~X" means the same as "~~X" — i.e., one literal tilde, and then one literal "X". However, by using "~X", you are assuming that no future version of Maketext will use "~X" as a magic escape sequence. In practice this is not a great problem, since first off you can just write "~~X" and not worry about it; second off, I doubt I'll add lots of new magic characters to bracket notation; and third off, you aren't likely to want literal "~" characters in your messages anyway, since it's not a character with wide use in natural language text.

Brackets must be balanced — every openbracket must have one matching closebracket, and vice versa. So these are all **invalid**:

```

"I ate [quant,_1,rhubarb pie."
"I ate [quant,_1,rhubarb pie[."
"I ate quant,_1,rhubarb pie] ."
"I ate quant,_1,rhubarb pie[."

```

Currently, bracket groups do not nest. That is, you **cannot** say:

```

"Foo [bar,baz,[quux,quuux]]\n";

```

If you need a notation that's that powerful, use normal Perl:

```

%Lexicon = (
    ...
    "some_key" => sub {
        my $lh = $_[0];
        join '',
            "Foo ",
            $lh->bar('baz', $lh->quux('quuux')),
            "\n",
        },
    ...
);

```

Or write the "bar" method so you don't need to pass it the output from calling quux.

I do not anticipate that you will need (or particularly want) to nest bracket groups, but you are welcome to email me with convincing (real-life) arguments to the contrary.

AUTO LEXICONS

If maketext goes to look in an individual %Lexicon for an entry for *key* (where *key* does not start with an underscore), and sees none, **but does see** an entry of "_AUTO" = *some_true_value*, then we actually define `$Lexicon{key} = key` right then and there, and then use that value as if it had been there all along. This happens before we even look in any superclass %Lexicons!

(This is meant to be somewhat like the AUTOLOAD mechanism in Perl's function call system — or, looked at another way, like the [AutoLoader/AutoLoader](#) module.)

I can picture all sorts of circumstances where you just do not want lookup to be able to fail (since failing normally means that maketext throws a die, altho see the next section for greater control over that). But here's one circumstance where _AUTO lexicons are meant to be *especially* useful:

As you're writing an application, you decide as you go what messages you need to emit. Normally you'd go to write this:

```
if(-e $filename) {
    go_process_file($filename)
} else {
    print "Couldn't find file \"$filename\"!\n";
}
```

but since you anticipate localizing this, you write:

```
use ThisProject::I18N;
my $lh = ThisProject::I18N->get_handle();
# For the moment, assume that things are set up so
# that we load class ThisProject::I18N::en
# and that's the class that $lh belongs to.
...
if(-e $filename) {
    go_process_file($filename)
} else {
    print $lh->maketext(
        "Couldn't find file \"[_1]\"!\n", $filename
    );
}
```

Now, right after you've just written the above lines, you'd normally have to go open the file `ThisProject/I18N/en.pm`, and immediately add an entry:

```
"Couldn't find file \"[_1]\"!\n"
=> "Couldn't find file \"[_1]\"!\n",
```

But I consider that somewhat of a distraction from the work of getting the main code working — to say nothing of the fact that I often have to play with the program a few times before I can decide exactly what wording I want in the messages (which in this case would require me to go changing three lines of code: the call to maketext with that key, and then the two lines in `ThisProject/I18N/en.pm`).

However, if you set "_AUTO = 1" in the %Lexicon in, `ThisProject/I18N/en.pm` (assuming that English (en) is the language that all your programmers will be using for this project's internal message keys), then you don't ever have to go adding lines like this

```
"Couldn't find file \"[_1]\"!\n"
=> "Couldn't find file \"[_1]\"!\n",
```

to ThisProject/I18N/en.pm, because if `_AUTO` is true there, then just looking for an entry with the key "Couldn't find file \"[_1]\"!\n" in that lexicon will cause it to be added, with that value!

Note that the reason that keys that start with `"_"` are immune to `_AUTO` isn't anything generally magical about the underscore character — I just wanted a way to have most lexicon keys be autoable, except for possibly a few, and I arbitrarily decided to use a leading underscore as a signal to distinguish those few.

CONTROLLING LOOKUP FAILURE

If you call `$lh->maketext(key, ...parameters...)`, and there's no entry *key* in `$lh`'s class's `%Lexicon`, nor in the superclass `%Lexicon` hash, *and* if we can't auto-make *key* (because either it starts with a `"_"`, or because none of its lexicons have `_AUTO => 1`), then we have failed to find a normal way to maketext *key*. What then happens in these failure conditions, depends on the `$lh` object "fail" attribute.

If the language handle has no "fail" attribute, maketext will simply throw an exception (i.e., it calls `die`, mentioning the *key* whose lookup failed, and naming the line number where the calling `$lh->maketext(key,...)` was.

If the language handle has a "fail" attribute whose value is a coderef, then `$lh->maketext(key,...params...)` gives up and calls:

```
return &{$that_subref}($lh, $key, @params);
```

Otherwise, the "fail" attribute's value should be a string denoting a method name, so that `$lh->maketext(key,...params...)` can give up with:

```
return $lh->$that_method_name($phrase, @params);
```

The "fail" attribute can be accessed with the `fail_with` method:

```
# Set to a coderef:
$lh->fail_with( \&failure_handler );

# Set to a method name:
$lh->fail_with( 'failure_method' );

# Set to nothing (i.e., so failure throws a plain exception)
$lh->fail_with( undef );

# Simply read:
$handler = $lh->fail_with();
```

Now, as to what you may want to do with these handlers: Maybe you'd want to log what key failed for what class, and then die. Maybe you don't like `die` and instead you want to send the error message to `STDOUT` (or wherever) and then merely `exit()`.

Or maybe you don't want to die at all! Maybe you could use a handler like this:

```
# Make all lookups fall back onto an English value,
# but after we log it for later fingerprinting.
my $lh_backup = ThisProject->get_handle('en');
open(LEX_FAIL_LOG, ">>wherever/lex.log") || die "GNAARGH $!";
sub lex_fail {
    my($failing_lh, $key, $params) = @_;
    print LEX_FAIL_LOG scalar(localtime), "\t",
        ref($failing_lh), "\t", $key, "\n";
    return $lh_backup->maketext($key,@params);
}
```

Some users have expressed that they think this whole mechanism of having a "fail" attribute at all, seems a rather pointless complication. But I want `Locale::Maketext` to be usable for software projects of *any* scale and type; and different software projects have different ideas of what the right thing is to do in failure conditions. I could simply say that failure always throws an exception, and that if you want to be careful,

you'll just have to wrap every call to `$lh->maketext` in an `eval { }`. However, I want programmers to reserve the right (via the "fail" attribute) to treat lookup failure as something other than an exception of the same level of severity as a config file being unreadable, or some essential resource being inaccessible.

One possibly useful value for the "fail" attribute is the method name `failure_handler_auto`. This is a method defined in class `Locale::Maketext` itself. You set it with:

```
$lh->fail_with('failure_handler_auto');
```

Then when you call `$lh->maketext(key, ...parameters...)` and there's no *key* in any of those lexicons, `maketext` gives up with

```
return $lh->failure_handler_auto($key, @params);
```

But `failure_handler_auto`, instead of dying or anything, compiles `$key`, caching it in `$lh->{'failure_lex'}{$key} = $compiled`, and then calls the compiled value, and returns that. (I.e., if `$key` looks like bracket notation, `$compiled` is a sub, and we return `&{$compiled}(@params)`; but if `$key` is just a plain string, we just return that.)

The effect of using `failure_handler_auto` is like an AUTO lexicon, except that it 1) compiles `$key` even if it starts with `"_"`, and 2) you have a record in the new hashref `$lh->{'failure_lex'}` of all the keys that have failed for this object. This should avoid your program dying — as long as your keys aren't actually invalid as bracket code, and as long as they don't try calling methods that don't exist.

`failure_handler_auto` may not be exactly what you want, but I hope it at least shows you that `maketext` failure can be mitigated in any number of very flexible ways. If you can formalize exactly what you want, you should be able to express that as a failure handler. You can even make it default for every object of a given class, by setting it in that class's `init`:

```
sub init {
    my $lh = $_[0]; # a newborn handle
    $lh->SUPER::init();
    $lh->fail_with('my_clever_failure_handler');
    return;
}
sub my_clever_failure_handler {
    ...you clever things here...
}
```

HOW TO USE MAKETEXT

Here is a brief checklist on how to use `Maketext` to localize applications:

- Decide what system you'll use for lexicon keys. If you insist, you can use opaque IDs (if you're nostalgic for `catgets`), but I have better suggestions in the section "Entries in Each Lexicon", above. Assuming you opt for meaningful keys that double as values (like "Minimum ([_1]) is larger than maximum ([_2])!n"), you'll have to settle on what language those should be in. For the sake of argument, I'll call this English, specifically American English, "en-US".
- Create a class for your localization project. This is the name of the class that you'll use in the idiom:

```
use Projname::L10N;
my $lh = Projname::L10N->get_handle(...) || die "Language?";
```

Assuming you call your class `Projname::L10N`, create a class consisting minimally of:

```
package Projname::L10N;
use base qw(Locale::Maketext);
...any methods you might want all your languages to share...

# And, assuming you want the base class to be an _AUTO lexicon,
# as is discussed a few sections up:
```

```
1;
```

- Create a class for the language your internal keys are in. Name the class after the language-tag for that language, in lowercase, with dashes changed to underscores. Assuming your project's first language is US English, you should call this `Projname::L10N::en_us`. It should consist minimally of:

```
package Projname::L10N::en_us;
use base qw(Projname::L10N);
%Lexicon = (
    '_AUTO' => 1,
);
1;
```

(For the rest of this section, I'll assume that this "first language class" of `Projname::L10N::en_us` has `_AUTO` lexicon.)

- Go and write your program. Everywhere in your program where you would say:

```
print "Foobar $thing stuff\n";
```

instead do it thru `maketext`, using no variable interpolation in the key:

```
print $lh->maketext("Foobar [_1] stuff\n", $thing);
```

If you get tired of constantly saying `print $lh->maketext`, consider making a functional wrapper for it, like so:

```
use Projname::L10N;
use vars qw($lh);
$lh = Projname::L10N->get_handle(...) || die "Language?";
sub pmt (@) { print( $lh->maketext(@_)) }
# "pmt" is short for "Print MakeText"
$Carp::Verbose = 1;
# so if maketext fails, we see made the call to pmt
```

Besides whole phrases meant for output, anything language-dependent should be put into the class `Projname::L10N::en_us`, whether as methods, or as lexicon entries — this is discussed in the section "Entries in Each Lexicon", above.

- Once the program is otherwise done, and once its localization for the first language works right (via the data and methods in `Projname::L10N::en_us`), you can get together the data for translation. If your first language lexicon isn't an `_AUTO` lexicon, then you already have all the messages explicitly in the lexicon (or else you'd be getting exceptions thrown when you call `$lh->maketext` to get messages that aren't in there). But if you were (advisedly) lazy and are using an `_AUTO` lexicon, then you've got to make a list of all the phrases that you've so far been letting `_AUTO` generate for you. There are very many ways to assemble such a list. The most straightforward is to simply `grep` the source for every occurrence of "maketext" (or calls to wrappers around it, like the above `pmt` function), and to log the following phrase.
- You may at this point want to consider whether the your base class (`Projname::L10N`) that all lexicons inherit from (`Projname::L10N::en`, `Projname::L10N::es`, etc.) should be an `_AUTO` lexicon. It may be true that in theory, all needed messages will be in each language class; but in the presumably unlikely or "impossible" case of lookup failure, you should consider whether your program should throw an exception, emit text in English (or whatever your project's first language is), or some more complex solution as described in the section "Controlling Lookup Failure", above.
- Submit all messages/phrases/etc. to translators.

(You may, in fact, want to start with localizing to *one* other language at first, if you're not sure that you've property abstracted the language-dependent parts of your code.)

Translators may request clarification of the situation in which a particular phrase is found. For example, in English we are entirely happy saying "*n* files found", regardless of whether we mean "I looked for files, and found *n* of them" or the rather distinct situation of "I looked for something else (like lines in files), and along the way I saw *n* files." This may involve rethinking things that you thought quite clear: should "Edit" on a toolbar be a noun ("editing") or a verb ("to edit")? Is there already a conventionalized way to express that menu option, separate from the target language's normal word for "to edit"?

In all cases where the very common phenomenon of quantification (saying "*N* files", for **any** value of *N*) is involved, each translator should make clear what dependencies the number causes in the sentence. In many cases, dependency is limited to words adjacent to the number, in places where you might expect them ("I found the-?PLURAL *N* empty-?PLURAL directory-?PLURAL"), but in some cases there are unexpected dependencies ("I found-?PLURAL ...") as well as long-distance dependencies "The *N* directory-?PLURAL could not be deleted-?PLURAL").

Remind the translators to consider the case where *N* is 0: "0 files found" isn't exactly natural-sounding in any language, but it may be unacceptable in many — or it may condition special kinds of agreement (similar to English "I didN'T find ANY files").

Remember to ask your translators about numeral formatting in their language, so that you can override the `numf` method as appropriate. Typical variables in number formatting are: what to use as a decimal point (comma? period?); what to use as a thousands separator (space? nonbreaking space? comma? period? small middot? prime? apostrophe?); and even whether the so-called "thousands separator" is actually for every third digit — I've heard reports of two hundred thousand being expressible as "2,00,000" for some Indian (Subcontinental) languages, besides the less surprising "200 000", "200.000", "200,000", and "200'000". Also, using a set of numeral glyphs other than the usual ASCII "0"–"9" might be appreciated, as via `tr/0-9/\x{0966}-\x{096F}/` for getting digits in Devanagari script (for Hindi, Konkani, others).

The basic `quant` method that `Locale::Maketext` provides should be good for many languages. For some languages, it might be useful to modify it (or its constituent `numerate` method) to take a plural form in the two-argument call to `quant` (as in "[quant,_1,files]") if it's all-around easier to infer the singular form from the plural, than to infer the plural form from the singular.

But for other languages (as is discussed at length in

[Locale::Maketext::TPJ13/Locale::Maketext::TPJ13](#)), simple `quant/numerify` is not enough. For the particularly problematic Slavic languages, what you may need is a method which you provide with the number, the citation form of the noun to quantify, and the case and gender that the sentence's syntax projects onto that noun slot. The method would then be responsible for determining what grammatical number that numeral projects onto its noun phrase, and what case and gender it may override the normal case and gender with; and then it would look up the noun in a lexicon providing all needed inflected forms.

- You may also wish to discuss with the translators the question of how to relate different subforms of the same language tag, considering how this reacts with `get_handle`'s treatment of these. For example, if a user accepts interfaces in "en, fr", and you have interfaces available in "en-US" and "fr", what should they get? You may wish to resolve this by establishing that "en" and "en-US" are effectively synonymous, by having one class zero-derive from the other.

For some languages this issue may never come up (Danish is rarely expressed as "da-DK", but instead is just "da"). And for other languages, the whole concept of a "generic" form may verge on being uselessly vague, particularly for interfaces involving voice media in forms of Arabic or Chinese.

- Once you've localized your program/site/etc. for all desired languages, be sure to show the result (whether live, or via screenshots) to the translators. Once they approve, make every effort to have it then checked by at least one other speaker of that language. This holds true even when (or especially when) the translation is done by one of your own programmers. Some kinds of systems may be harder to find testers for than others, depending on the amount of domain-specific jargon and concepts

involved — it's easier to find people who can tell you whether they approve of your translation for "delete this message" in an email-via-Web interface, than to find people who can give you an informed opinion on your translation for "attribute value" in an XML query tool's interface.

SEE ALSO

I recommend reading all of these:

[Locale::Maketext::TPJ13/Locale::Maketext::TPJ13](#) — my *The Perl Journal* article about Maketext. It explains many important concepts underlying Locale::Maketext's design, and some insight into why Maketext is better than the plain old approach of just having message catalogs that are just databases of printf formats.

[File::Findgrep/File::Findgrep](#) is a sample application/module that uses Locale::Maketext to localize its messages.

[I18N::LangTags/I18N::LangTags](#).

[Win32::Locale/Win32::Locale](#).

RFC 3066, *Tags for the Identification of Languages*, as at <http://sunsite.dk/RFC/rfc/rfc3066.html>

RFC 2277, *IETF Policy on Character Sets and Languages* is at <http://sunsite.dk/RFC/rfc/rfc2277.html> — much of it is just things of interest to protocol designers, but it explains some basic concepts, like the distinction between locales and language-tags.

The manual for GNU gettext. The gettext dist is available in <ftp://prep.ai.mit.edu/pub/gnu/> — get a recent gettext tarball and look in its "doc/" directory, there's an easily browsable HTML version in there. The gettext documentation asks lots of questions worth thinking about, even if some of their answers are sometimes wonky, particularly where they start talking about pluralization.

The Locale/Maketext.pm source. Observe that the module is much shorter than its documentation!

COPYRIGHT AND DISCLAIMER

Copyright (c) 1999–2001 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Sean M. Burke sburke@cpan.org

NAME

Locale::Maketext::TPJ13 — article about software localization

SYNOPSIS

```
# This an article, not a module.
```

DESCRIPTION

The following article by Sean M. Burke and Jordan Lachler first appeared in *The Perl Journal* #13 and is copyright 1999 The Perl Journal. It appears courtesy of Jon Orwant and The Perl Journal. This document may be distributed under the same terms as Perl itself.

Localization and Perl: gettext breaks, Maketext fixes

by Sean M. Burke and Jordan Lachler

This article points out cases where gettext (a common system for localizing software interfaces — i.e., making them work in the user's language of choice) fails because of basic differences between human languages. This article then describes Maketext, a new system capable of correctly treating these differences.

A Localization Horror Story: It Could Happen To You

"There are a number of languages spoken by human beings in this world."

— Harald Tveit Alvestrand, in RFC 1766, "Tags for the Identification of Languages"

Imagine that your task for the day is to localize a piece of software — and luckily for you, the only output the program emits is two messages, like this:

```
I scanned 12 directories.
```

```
Your query matched 10 files in 4 directories.
```

So how hard could that be? You look at the code that produces the first item, and it reads:

```
printf("I scanned %g directories.",
      $directory_count);
```

You think about that, and realize that it doesn't even work right for English, as it can produce this output:

```
I scanned 1 directories.
```

So you rewrite it to read:

```
printf("I scanned %g %s.",
      $directory_count,
      $directory_count == 1 ?
        "directory" : "directories",
    );
```

...which does the Right Thing. (In case you don't recall, "%g" is for locale-specific number interpolation, and "%s" is for string interpolation.)

But you still have to localize it for all the languages you're producing this software for, so you pull Locale::gettext off of CPAN so you can access the gettext C functions you've heard are standard for localization tasks.

And you write:

```
printf(gettext("I scanned %g %s."),
      $dir_scan_count,
      $dir_scan_count == 1 ?
        gettext("directory") : gettext("directories"),
    );
```

But you then read in the gettext manual (Drepper, Miller, and Pinard 1995) that this is not a good idea, since how a single word like "directory" or "directories" is translated may depend on context — and this is true, since in a case language like German or Russian, you'd may need these words with a different case ending in the first instance (where the word is the object of a verb) than in the second instance, which you haven't even gotten to yet (where the word is the object of a preposition, "in %g directories") — assuming these keep the same syntax when translated into those languages.

So, on the advice of the gettext manual, you rewrite:

```
printf( $dir_scan_count == 1 ?
        gettext("I scanned %g directory.") :
        gettext("I scanned %g directories."),
        $dir_scan_count );
```

So, you email your various translators (the boss decides that the languages du jour are Chinese, Arabic, Russian, and Italian, so you have one translator for each), asking for translations for "I scanned %g directory." and "I scanned %g directories.". When they reply, you'll put that in the lexicons for gettext to use when it localizes your software, so that when the user is running under the "zh" (Chinese) locale, gettext("I scanned %g directory.") will return the appropriate Chinese text, with a "%g" in there where printf can then interpolate \$dir_scan.

Your Chinese translator emails right back — he says both of these phrases translate to the same thing in Chinese, because, in linguistic jargon, Chinese "doesn't have number as a grammatical category" — whereas English does. That is, English has grammatical rules that refer to "number", i.e., whether something is grammatically singular or plural; and one of these rules is the one that forces nouns to take a plural suffix (generally "s") when in a plural context, as they are when they follow a number other than "one" (including, oddly enough, "zero"). Chinese has no such rules, and so has just the one phrase where English has two. But, no problem, you can have this one Chinese phrase appear as the translation for the two English phrases in the "zh" gettext lexicon for your program.

Emboldened by this, you dive into the second phrase that your software needs to output: "Your query matched 10 files in 4 directories.". You notice that if you want to treat phrases as indivisible, as the gettext manual wisely advises, you need four cases now, instead of two, to cover the permutations of singular and plural on the two items, \$dir_count and \$file_count. So you try this:

```
printf( $file_count == 1 ?
        ( $directory_count == 1 ?
          gettext("Your query matched %g file in %g directory.") :
          gettext("Your query matched %g file in %g directories.") ) :
        ( $directory_count == 1 ?
          gettext("Your query matched %g files in %g directory.") :
          gettext("Your query matched %g files in %g directories.") ),
        $file_count, $directory_count,
        );
```

(The case of "1 file in 2 [or more] directories" could, I suppose, occur in the case of symlinking or something of the sort.)

It occurs to you that this is not the prettiest code you've ever written, but this seems the way to go. You mail off to the translators asking for translations for these four cases. The Chinese guy replies with the one phrase that these all translate to in Chinese, and that phrase has two "%g"s in it, as it should — but there's a problem. He translates it word-for-word back: "In %g directories contains %g files match your query." The %g slots are in an order reverse to what they are in English. You wonder how you'll get gettext to handle that.

But you put it aside for the moment, and optimistically hope that the other translators won't have this problem, and that their languages will be better behaved — i.e., that they will be just like English.

But the Arabic translator is the next to write back. First off, your code for "I scanned %g directory." or "I

scanned %g directories." assumes there's only singular or plural. But, to use linguistic jargon again, Arabic has grammatical number, like English (but unlike Chinese), but it's a three-term category: singular, dual, and plural. In other words, the way you say "directory" depends on whether there's one directory, or *two* of them, or *more than two* of them. Your test of (`$directory == 1`) no longer does the job. And it means that where English's grammatical category of number necessitates only the two permutations of the first sentence based on "directory [singular]" and "directories [plural]", Arabic has three — and, worse, in the second sentence ("Your query matched %g file in %g directory."), where English has four, Arabic has nine. You sense an unwelcome, exponential trend taking shape.

Your Italian translator emails you back and says that "I searched 0 directories" (a possible English output of your program) is stilted, and if you think that's fine English, that's your problem, but that *just will not do* in the language of Dante. He insists that where `$directory_count` is 0, your program should produce the Italian text for "I *didn't* scan *any* directories.". And ditto for "I didn't match any files in any directories", although he says the last part about "in any directories" should probably just be left off.

You wonder how you'll get `gettext` to handle this; to accomodate the ways Arabic, Chinese, and Italian deal with numbers in just these few very simple phrases, you need to write code that will ask `gettext` for different queries depending on whether the numerical values in question are 1, 2, more than 2, or in some cases 0, and you still haven't figured out the problem with the different word order in Chinese.

Then your Russian translator calls on the phone, to *personally* tell you the bad news about how really unpleasant your life is about to become:

Russian, like German or Latin, is an inflectional language; that is, nouns and adjectives have to take endings that depend on their case (i.e., nominative, accusative, genitive, etc...) — which is roughly a matter of what role they have in syntax of the sentence — as well as on the grammatical gender (i.e., masculine, feminine, neuter) and number (i.e., singular or plural) of the noun, as well as on the declension class of the noun. But unlike with most other inflected languages, putting a number-phrase (like "ten" or "forty-three", or their Arabic numeral equivalents) in front of noun in Russian can change the case and number that noun is, and therefore the endings you have to put on it.

He elaborates: In "I scanned %g directories", you'd *expect* "directories" to be in the accusative case (since it is the direct object in the sentence) and the plural number, except where `$directory_count` is 1, then you'd expect the singular, of course. Just like Latin or German. *But!* Where `$directory_count % 10` is 1 ("`%`" for modulo, remember), assuming `$directory_count` is an integer, and except where `$directory_count % 100` is 11, "directories" is forced to become grammatically singular, which means it gets the ending for the accusative singular... You begin to visualize the code it'd take to test for the problem so far, *and still work for Chinese and Arabic and Italian*, and how many `gettext` items that'd take, but he keeps going... But where `$directory_count % 10` is 2, 3, or 4 (except where `$directory_count % 100` is 12, 13, or 14), the word for "directories" is forced to be genitive singular — which means another ending... The room begins to spin around you, slowly at first... But with *all other* integer values, since "directory" is an inanimate noun, when preceded by a number and in the nominative or accusative cases (as it is here, just your luck!), it does stay plural, but it is forced into the genitive case — yet another ending... And you never hear him get to the part about how you're going to run into similar (but maybe subtly different) problems with other Slavic languages like Polish, because the floor comes up to meet you, and you fade into unconsciousness.

The above cautionary tale relates how an attempt at localization can lead from programmer consternation, to program obfuscation, to a need for sedation. But careful evaluation shows that your choice of tools merely needed further consideration.

The Linguistic View

"It is more complicated than you think."

— The Eighth Networking Truth, from RFC 1925

The field of Linguistics has expended a great deal of effort over the past century trying to find grammatical patterns which hold across languages; it's been a constant process of people making generalizations that should apply to all languages, only to find out that, all too often, these generalizations fail — sometimes

failing for just a few languages, sometimes whole classes of languages, and sometimes nearly every language in the world except English. Broad statistical trends are evident in what the "average language" is like as far as what its rules can look like, must look like, and cannot look like. But the "average language" is just as unreal a concept as the "average person" — it runs up against the fact no language (or person) is, in fact, average. The wisdom of past experience leads us to believe that any given language can do whatever it wants, in any order, with appeal to any kind of grammatical categories wants — case, number, tense, real or metaphoric characteristics of the things that words refer to, arbitrary or predictable classifications of words based on what endings or prefixes they can take, degree or means of certainty about the truth of statements expressed, and so on, ad infinitum.

Mercifully, most localization tasks are a matter of finding ways to translate whole phrases, generally sentences, where the context is relatively set, and where the only variation in content is *usually* in a number being expressed — as in the example sentences above. Translating specific, fully-formed sentences is, in practice, fairly foolproof — which is good, because that's what's in the phrasebooks that so many tourists rely on. Now, a given phrase (whether in a phrasebook or in a gettext lexicon) in one language *might* have a greater or lesser applicability than that phrase's translation into another language — for example, strictly speaking, in Arabic, the "your" in "Your query matched..." would take a different form depending on whether the user is male or female; so the Arabic translation "your[feminine] query" is applicable in fewer cases than the corresponding English phrase, which doesn't distinguish the user's gender. (In practice, it's not feasible to have a program know the user's gender, so the masculine "you" in Arabic is usually used, by default.)

But in general, such surprises are rare when entire sentences are being translated, especially when the functional context is restricted to that of a computer interacting with a user either to convey a fact or to prompt for a piece of information. So, for purposes of localization, translation by phrase (generally by sentence) is both the simplest and the least problematic.

Breaking gettext

"It Has To Work."

— First Networking Truth, RFC 1925

Consider that sentences in a tourist phrasebook are of two types: ones like "How do I get to the marketplace?" that don't have any blanks to fill in, and ones like "How much do these ___ cost?", where there's one or more blanks to fill in (and these are usually linked to a list of words that you can put in that blank: "fish", "potatoes", "tomatoes", etc.) The ones with no blanks are no problem, but the fill-in-the-blank ones may not be really straightforward. If it's a Swahili phrasebook, for example, the authors probably didn't bother to tell you the complicated ways that the verb "cost" changes its inflectional prefix depending on the noun you're putting in the blank. The trader in the marketplace will still understand what you're saying if you say "how much do these potatoes cost?" with the wrong inflectional prefix on "cost". After all, *you* can't speak proper Swahili, *you're* just a tourist. But while tourists can be stupid, computers are supposed to be smart; the computer should be able to fill in the blank, and still have the results be grammatical.

In other words, a phrasebook entry takes some values as parameters (the things that you fill in the blank or blanks), and provides a value based on these parameters, where the way you get that final value from the given values can, properly speaking, involve an arbitrarily complex series of operations. (In the case of Chinese, it'd be not at all complex, at least in cases like the examples at the beginning of this article; whereas in the case of Russian it'd be a rather complex series of operations. And in some languages, the complexity could be spread around differently: while the act of putting a number-expression in front of a noun phrase might not be complex by itself, it may change how you have to, for example, inflect a verb elsewhere in the sentence. This is what in syntax is called "long-distance dependencies".)

This talk of parameters and arbitrary complexity is just another way to say that an entry in a phrasebook is what in a programming language would be called a "function". Just so you don't miss it, this is the crux of this article: *A phrase is a function; a phrasebook is a bunch of functions.*

The reason that using gettext runs into walls (as in the above second-person horror story) is that you're

trying to use a string (or worse, a choice among a bunch of strings) to do what you really need a function for — which is futile. Preforming (s)printf interpolation on the strings which you get back from gettext does allow you to do *some* common things passably well... sometimes... sort of; but, to paraphrase what some people say about `csh` script programming, "it fools you into thinking you can use it for real things, but you can't, and you don't discover this until you've already spent too much time trying, and by then it's too late."

Replacing gettext

So, what needs to replace gettext is a system that supports lexicons of functions instead of lexicons of strings. An entry in a lexicon from such a system should *not* look like this:

```
"J'ai trouv\xE9 %g fichiers dans %g r\xE9pertoires"
```

[`\xE9` is e-acute in Latin-1. Some pod renderers would scream if I used the actual character here. — SB]

but instead like this, bearing in mind that this is just a first stab:

```
sub I_found_X1_files_in_X2_directories {
    my( $files, $dirs ) = @_ [0,1];
    $files = sprintf("%g %s", $files,
        $files == 1 ? 'fichier' : 'fichiers');
    $dirs = sprintf("%g %s", $dirs,
        $dirs == 1 ? "r\xE9pertoire" : "r\xE9pertoires");
    return "J'ai trouv\xE9 $files dans $dirs.";
}
```

Now, there's no particularly obvious way to store anything but strings in a gettext lexicon; so it looks like we just have to start over and make something better, from scratch. I call my shot at a gettext-replacement system "Maketext", or, in CPAN terms, `Locale::Maketext`.

When designing Maketext, I chose to plan its main features in terms of "buzzword compliance". And here are the buzzwords:

Buzzwords: Abstraction and Encapsulation

The complexity of the language you're trying to output a phrase in is entirely abstracted inside (and encapsulated within) the Maketext module for that interface. When you call:

```
print $lang->maketext("You have [quant,_1,piece] of new mail.",
    scalar(@messages));
```

you don't know (and in fact can't easily find out) whether this will involve lots of figuring, as in Russian (if `$lang` is a handle to the Russian module), or relatively little, as in Chinese. That kind of abstraction and encapsulation may encourage other pleasant buzzwords like modularization and stratification, depending on what design decisions you make.

Buzzword: Isomorphism

"Isomorphism" means "having the same structure or form"; in discussions of program design, the word takes on the special, specific meaning that your implementation of a solution to a problem *has the same structure* as, say, an informal verbal description of the solution, or maybe of the problem itself. Isomorphism is, all things considered, a good thing — it's what problem-solving (and solution-implementing) should look like.

What's wrong the with gettext—using code like this...

```
printf( $file_count == 1 ?
    ( $directory_count == 1 ?
        "Your query matched %g file in %g directory." :
        "Your query matched %g file in %g directories." ) :
    ( $directory_count == 1 ?
        "Your query matched %g files in %g directory." :
        "Your query matched %g files in %g directories." ),
    $file_count, $directory_count,
```

);

is first off that it's not well abstracted — these ways of testing for grammatical number (as in the expressions like `foo == 1 ? singular_form : plural_form`) should be abstracted to each language module, since how you get grammatical number is language-specific.

But second off, it's not isomorphic — the "solution" (i.e., the phrasebook entries) for Chinese maps from these four English phrases to the one Chinese phrase that fits for all of them. In other words, the informal solution would be "The way to say what you want in Chinese is with the one phrase 'For your question, in Y directories you would find X files'" — and so the implemented solution should be, isomorphically, just a straightforward way to spit out that one phrase, with numerals properly interpolated. It shouldn't have to map from the complexity of other languages to the simplicity of this one.

Buzzword: Inheritance

There's a great deal of reuse possible for sharing of phrases between modules for related dialects, or for sharing of auxiliary functions between related languages. (By "auxiliary functions", I mean functions that don't produce phrase-text, but which, say, return an answer to "does this number require a plural noun after it?". Such auxiliary functions would be used in the internal logic of functions that actually do produce phrase-text.)

In the case of sharing phrases, consider that you have an interface already localized for American English (probably by having been written with that as the native locale, but that's incidental). Localizing it for UK English should, in practical terms, be just a matter of running it past a British person with the instructions to indicate what few phrases would benefit from a change in spelling or possibly minor rewording. In that case, you should be able to put in the UK English localization module *only* those phrases that are UK-specific, and for all the rest, *inherit* from the American English module. (And I expect this same situation would apply with Brazilian and Continental Portuguese, possibly with some *very* closely related languages like Czech and Slovak, and possibly with the slightly different "versions" of written Mandarin Chinese, as I hear exist in Taiwan and mainland China.)

As to sharing of auxiliary functions, consider the problem of Russian numbers from the beginning of this article; obviously, you'd want to write only once the hairy code that, given a numeric value, would return some specification of which case and number a given quantified noun should use. But suppose that you discover, while localizing an interface for, say, Ukrainian (a Slavic language related to Russian, spoken by several million people, many of whom would be relieved to find that your Web site's or software's interface is available in their language), that the rules in Ukrainian are the same as in Russian for quantification, and probably for many other grammatical functions. While there may well be no phrases in common between Russian and Ukrainian, you could still choose to have the Ukrainian module inherit from the Russian module, just for the sake of inheriting all the various grammatical methods. Or, probably better organizationally, you could move those functions to a module called `_E_Slavic` or something, which Russian and Ukrainian could inherit useful functions from, but which would (presumably) provide no lexicon.

Buzzword: Concision

Okay, concision isn't a buzzword. But it should be, so I decree that as a new buzzword, "concision" means that simple common things should be expressible in very few lines (or maybe even just a few characters) of code — call it a special case of "making simple things easy and hard things possible", and see also the role it played in the MIDI::Simple language, discussed elsewhere in this issue [TPJ#13].

Consider our first stab at an entry in our "phrasebook of functions":

```
sub I_found_X1_files_in_X2_directories {
    my( $files, $dirs ) = @_ [0,1];
    $files = sprintf("%g %s", $files,
        $files == 1 ? 'fichier' : 'fichiers');
    $dirs = sprintf("%g %s", $dirs,
        $dirs == 1 ? "r\xE9pertoire" : "r\xE9pertoires");
    return "J'ai trouv\xE9 $files dans $dirs.";
}
```

You may sense that a lexicon (to use a non-committal catch-all term for a collection of things you know how to say, regardless of whether they're phrases or words) consisting of functions *expressed* as above would make for rather long-winded and repetitive code — even if you wisely rewrote this to have quantification (as we call adding a number expression to a noun phrase) be a function called like:

```
sub I_found_X1_files_in_X2_directories {
    my( $files, $dirs ) = @_ [0,1];
    $files = quant($files, "fichier");
    $dirs = quant($dirs, "r\xE9pertoire");
    return "J'ai trouv\xE9 $files dans $dirs.";
}
```

And you may also sense that you do not want to bother your translators with having to write Perl code — you'd much rather that they spend their *very costly time* on just translation. And this is to say nothing of the near impossibility of finding a commercial translator who would know even simple Perl.

In a first-hack implementation of Maketext, each language-module's lexicon looked like this:

```
%Lexicon = (
    "I found %g files in %g directories"
    => sub {
        my( $files, $dirs ) = @_ [0,1];
        $files = quant($files, "fichier");
        $dirs = quant($dirs, "r\xE9pertoire");
        return "J'ai trouv\xE9 $files dans $dirs.";
    },
    ... and so on with other phrase => sub mappings ...
);
```

but I immediately went looking for some more concise way to basically denote the same phrase-function — a way that would also serve to concisely denote *most* phrase-functions in the lexicon for *most* languages. After much time and even some actual thought, I decided on this system:

- * Where a value in a %Lexicon hash is a contentful string instead of an anonymous sub (or, conceivably, a coderef), it would be interpreted as a sort of shorthand expression of what the sub does. When accessed for the first time in a session, it is parsed, turned into Perl code, and then eval'd into an anonymous sub; then that sub replaces the original string in that lexicon. (That way, the work of parsing and evaluating the shorthand form for a given phrase is done no more than once per session.)

- * Calls to maketext (as Maketext's main function is called) happen thru a "language session handle", notionally very much like an IO handle, in that you open one at the start of the session, and use it for "sending signals" to an object in order to have it return the text you want.

So, this:

```
$lang->maketext("You have [quant,_1,piece] of new mail.",
               scalar(@messages));
```

basically means this: look in the lexicon for \$lang (which may inherit from any number of other lexicons), and find the function that we happen to associate with the string "You have [quant,_1,piece] of new mail" (which is, and should be, a functioning "shorthand" for this function in the native locale — English in this case). If you find such a function, call it with \$lang as its first parameter (as if it were a method), and then a copy of scalar(@messages) as its second, and then return that value. If that function was found, but was in string shorthand instead of being a fully specified function, parse it and make it into a function before calling it the first time.

- * The shorthand uses code in brackets to indicate method calls that should be performed. A full explanation is not in order here, but a few examples will suffice:

```
"You have [quant,_1,piece] of new mail."
```


The above code is shorthand for, and will be interpreted as, this:

```
sub {
    my $handle = $_[0];
    my(@params) = @_;
    return join '',
        "You have ",
        $handle->quant($params[1], 'piece'),
        "of new mail.";
}
```

where "quant" is the name of a method you're using to quantify the noun "piece" with the number \$params[0].

A string with no bracket calls, like this:

```
"Your search expression was malformed."
```

is somewhat of a degerate case, and just gets turned into:

```
sub { return "Your search expression was malformed." }
```

However, not everything you can write in Perl code can be written in the above shorthand system — not by a long shot. For example, consider the Italian translator from the beginning of this article, who wanted the Italian for "I didn't find any files" as a special case, instead of "I found 0 files". That couldn't be specified (at least not easily or simply) in our shorthand system, and it would have to be written out in full, like this:

```
sub { # pretend the English strings are in Italian
    my($handle, $files, $dirs) = @_[0,1,2];
    return "I didn't find any files" unless $files;
    return join '',
        "I found ",
        $handle->quant($files, 'file'),
        " in ",
        $handle->quant($dirs, 'directory'),
        ".";
}
```

Next to a lexicon full of shorthand code, that sort of sticks out like a sore thumb — but this *is* a special case, after all; and at least it's possible, if not as concise as usual.

As to how you'd implement the Russian example from the beginning of the article, well, There's More Than One Way To Do It, but it could be something like this (using English words for Russian, just so you know what's going on):

```
"I [quant,_1,directory,accusative] scanned."
```

This shifts the burden of complexity off to the quant method. That method's parameters are: the numeric value it's going to use to quantify something; the Russian word it's going to quantify; and the parameter "accusative", which you're using to mean that this sentence's syntax wants a noun in the accusative case there, although that quantification method may have to overrule, for grammatical reasons you may recall from the beginning of this article.

Now, the Russian quant method here is responsible not only for implementing the strange logic necessary for figuring out how Russian number-phrases impose case and number on their noun-phrases, but also for inflecting the Russian word for "directory". How that inflection is to be carried out is no small issue, and among the solutions I've seen, some (like variations on a simple lookup in a hash where all possible forms are provided for all necessary words) are straightforward but *can* become cumbersome when you need to inflect more than a few dozen words; and other solutions (like using algorithms to model the inflections, storing only root forms and irregularities) *can* involve more overhead than is justifiable for all but the largest lexicons.

Mercifully, this design decision becomes crucial only in the hairiest of inflected languages, of which Russian is by no means the *worst* case scenario, but is worse than most. Most languages have simpler inflection systems; for example, in English or Swahili, there are generally no more than two possible inflected forms for a given noun ("error/errors"; "kosa/makosa"), and the rules for producing these forms are fairly simple — or at least, simple rules can be formulated that work for most words, and you can then treat the exceptions as just "irregular", at least relative to your ad hoc rules. A simpler inflection system (simpler rules, fewer forms) means that design decisions are less crucial to maintaining sanity, whereas the same decisions could incur overhead—versus—scalability problems in languages like Russian. It may *also* be likely that code (possibly in Perl, as with `Lingua::EN::Inflect`, for English nouns) has already been written for the language in question, whether simple or complex.

Moreover, a third possibility may even be simpler than anything discussed above: "Just require that all possible (or at least applicable) forms be provided in the call to the given language's `quant` method, as in:"

```
"I found [quant,_1,file,files]."
```

That way, `quant` just has to choose which form it needs, without having to look up or generate anything. While possibly not optimal for Russian, this should work well for most other languages, where quantification is not as complicated an operation.

The Devil in the Details

There's plenty more to `Maketext` than described above — for example, there's the details of how language tags ("en-US", "i-pwn", "fi", etc.) or locale IDs ("en_US") interact with actual module naming ("BogoQuery/Locale/en_us.pm"), and what magic can ensue; there's the details of how to record (and possibly negotiate) what character encoding `Maketext` will return text in (UTF8? Latin-1? KOI8?). There's the interesting fact that `Maketext` is for localization, but nowhere actually has a `"use locale;"` anywhere in it. For the curious, there's the somewhat frightening details of how I actually implement something like data inheritance so that searches across modules' `%Lexicon` hashes can parallel how Perl implements method inheritance.

And, most importantly, there's all the practical details of how to actually go about deriving from `Maketext` so you can use it for your interfaces, and the various tools and conventions for starting out and maintaining individual language modules.

That is all covered in the documentation for `Locale::Maketext` and the modules that come with it, available in CPAN. After having read this article, which covers the why's of `Maketext`, the documentation, which covers the how's of it, should be quite straightforward.

The Proof in the Pudding: Localizing Web Sites

`Maketext` and `gettext` have a notable difference: `gettext` is in C, accessible thru C library calls, whereas `Maketext` is in Perl, and really can't work without a Perl interpreter (although I suppose something like it could be written for C). Accidents of history (and not necessarily lucky ones) have made C++ the most common language for the implementation of applications like word processors, Web browsers, and even many in-house applications like custom query systems. Current conditions make it somewhat unlikely that the next one of any of these kinds of applications will be written in Perl, albeit clearly more for reasons of custom and inertia than out of consideration of what is the right tool for the job.

However, other accidents of history have made Perl a well-accepted language for design of server-side programs (generally in CGI form) for Web site interfaces. Localization of static pages in Web sites is trivial, feasible either with simple language-negotiation features in servers like Apache, or with some kind of server-side inclusions of language-appropriate text into layout templates. However, I think that the localization of Perl-based search systems (or other kinds of dynamic content) in Web sites, be they public or access-restricted, is where `Maketext` will see the greatest use.

I presume that it would be only the exceptional Web site that gets localized for English *and* Chinese *and* Italian *and* Arabic *and* Russian, to recall the languages from the beginning of this article — to say nothing of German, Spanish, French, Japanese, Finnish, and Hindi, to name a few languages that benefit from large numbers of programmers or Web viewers or both.

However, the ever-increasing internationalization of the Web (whether measured in terms of amount of content, of numbers of content writers or programmers, or of size of content audiences) makes it increasingly likely that the interface to the average Web-based dynamic content service will be localized for two or maybe three languages. It is my hope that Maketext will make that task as simple as possible, and will remove previous barriers to localization for languages dissimilar to English.

__END__

Sean M. Burke (sburkecpan.org) has a Master's in linguistics from Northwestern University; he specializes in language technology. Jordan Lachler (lachlerunm.edu) is a PhD student in the Department of Linguistics at the University of New Mexico; he specializes in morphology and pedagogy of North American native languages.

References

Alvestrand, Harald Tveit. 1995. *RFC 1766: Tags for the Identification of Languages*.
<ftp://ftp.isi.edu/in-notes/rfc1766.txt> [Now see RFC 3066.]

Callon, Ross, editor. 1996. *RFC 1925: The Twelve Networking Truths*.
<ftp://ftp.isi.edu/in-notes/rfc1925.txt>

Drepper, Ulrich, Peter Miller, and François Pinard. 1995–2001. GNU `gettext`. Available in <ftp://prep.ai.mit.edu/pub/gnu/>, with extensive docs in the distribution tarball. [Since I wrote this article in 1998, I now see that the `gettext` docs are now trying more to come to terms with plurality. Whether useful conclusions have come from it is another question altogether. — SMB, May 2001]

Forbes, Nevill. 1964. *Russian Grammar*. Third Edition, revised by J. C. Dumbreck. Oxford University Press.

NAME

Locale::Script – ISO codes for script identification (ISO 15924)

SYNOPSIS

```
use Locale::Script;
use Locale::Constants;

$script = code2script('ph');           # 'Phoenician'
$code   = script2code('Tibetan');      # 'bo'
$code3  = script2code('Tibetan',
                      LOCALE_CODE_ALPHA_3); # 'bod'
$codeN  = script2code('Tibetan',
                      LOCALE_CODE_ALPHA_NUMERIC); # 330

@codes  = all_script_codes();
@scripts = all_script_names();
```

DESCRIPTION

The `Locale::Script` module provides access to the ISO codes for identifying scripts, as defined in ISO 15924. For example, Egyptian hieroglyphs are denoted by the two-letter code 'eg', the three-letter code 'egy', and the numeric code 050.

You can either access the codes via the conversion routines (described below), or with the two functions which return lists of all script codes or all script names.

There are three different code sets you can use for identifying scripts:

alpha-2

Two letter codes, such as 'bo' for Tibetan. This code set is identified with the symbol `LOCALE_CODE_ALPHA_2`.

alpha-3

Three letter codes, such as 'ell' for Greek. This code set is identified with the symbol `LOCALE_CODE_ALPHA_3`.

numeric

Numeric codes, such as 410 for Hiragana. This code set is identified with the symbol `LOCALE_CODE_NUMERIC`.

All of the routines take an optional additional argument which specifies the code set to use. If not specified, it defaults to the two-letter codes. This is partly for backwards compatibility (previous versions of `Locale` modules only supported the alpha-2 codes), and partly because they are the most widely used codes.

The alpha-2 and alpha-3 codes are not case-dependent, so you can use 'BO', 'Bo', 'bO' or 'bo' for Tibetan. When a code is returned by one of the functions in this module, it will always be lower-case.

SPECIAL CODES

The standard defines various special codes.

- The standard reserves codes in the ranges **qa – qt**, **qaa – qat**, and **900 – 919**, for private use.
- **zx**, **zxx**, and **997**, are the codes for unwritten languages.
- **zy**, **zyy**, and **998**, are the codes for an undetermined script.
- **zz**, **zzz**, and **999**, are the codes for an uncoded script.

The private codes are not recognised by `Locale::Script`, but the others are.

CONVERSION ROUTINES

There are three conversion routines: `code2script()`, `script2code()`, and `script_code2code()`.

`code2script(CODE, [CODESET])`

This function takes a script code and returns a string which contains the name of the script identified. If the code is not a valid script code, as defined by ISO 15924, then `undef` will be returned:

```
$script = code2script('cy');    # Cyrillic
```

`script2code(STRING, [CODESET])`

This function takes a script name and returns the corresponding script code, if such exists. If the argument could not be identified as a script name, then `undef` will be returned:

```
$code = script2code('Gothic', LOCALE_CODE_ALPHA_3);
# $code will now be 'gth'
```

The case of the script name is not important. See the section [KNOWN BUGS AND LIMITATIONS](#) below.

`script_code2code(CODE, CODESET, CODESET)`

This function takes a script code from one code set, and returns the corresponding code from another code set.

```
$alpha2 = script_code2code('jwi',
                           LOCALE_CODE_ALPHA_3 => LOCALE_CODE_ALPHA_2);
# $alpha2 will now be 'jw' (Javanese)
```

If the code passed is not a valid script code in the first code set, or if there isn't a code for the corresponding script in the second code set, then `undef` will be returned.

QUERY ROUTINES

There are two function which can be used to obtain a list of all codes, or all script names:

`all_script_codes ([CODESET])`

Returns a list of all two-letter script codes. The codes are guaranteed to be all lower-case, and not in any particular order.

`all_script_names ([CODESET])`

Returns a list of all script names for which there is a corresponding script code in the specified code set. The names are capitalised, and not returned in any particular order.

EXAMPLES

The following example illustrates use of the `code2script()` function. The user is prompted for a script code, and then told the corresponding script name:

```
$| = 1;    # turn off buffering

print "Enter script code: ";
chop($code = <STDIN>);
$script = code2script($code, LOCALE_CODE_ALPHA_2);
if (defined $script)
{
    print "$code = $script\n";
}
else
{
    print "'$code' is not a valid script code!\n";
}
```

KNOWN BUGS AND LIMITATIONS

- When using `script2code()`, the script name must currently appear exactly as it does in the source of the module. For example,

```
script2code('Egyptian hieroglyphs')
```

will return **eg**, as expected. But the following will all return `undef`:

```
script2code('hieroglyphs')
script2code('Egyptian Hieroglyphics')
```

If there's need for it, a future version could have variants for script names.

- In the current implementation, all data is read in when the module is loaded, and then held in memory. A lazy implementation would be more memory friendly.

SEE ALSO

`Locale::Language`

ISO two letter codes for identification of language (ISO 639).

`Locale::Currency`

ISO three letter codes for identification of currencies and funds (ISO 4217).

`Locale::Country`

ISO three letter codes for identification of countries (ISO 3166)

ISO 15924

The ISO standard which defines these codes.

<http://www.evertype.com/standards/iso15924/>

Home page for ISO 15924.

AUTHOR

Neil Bowers <neil@bowers.com>

COPYRIGHT

Copyright (c) 2002 Neil Bowers.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

NAME

Math::BigFloat – Arbitrary size floating point math package

SYNOPSIS

```
use Math::BigFloat;

# Number creation
$x = Math::BigFloat->new($str);           # defaults to 0
$nan = Math::BigFloat->bnan();             # create a NaN
$zero = Math::BigFloat->bzero();           # create a +0
$inf = Math::BigFloat->binf();             # create a +inf
$inf = Math::BigFloat->binf('-');          # create a -inf
$one = Math::BigFloat->bone();              # create a +1
$one = Math::BigFloat->bone('-');          # create a -1

# Testing
$x->is_zero();                             # true if arg is +0
$x->is_nan();                              # true if arg is NaN
$x->is_one();                              # true if arg is +1
$x->is_one('-');                           # true if arg is -1
$x->is_odd();                              # true if odd, false for even
$x->is_even();                             # true if even, false for odd
$x->is_positive();                         # true if >= 0
$x->is_negative();                        # true if < 0
$x->is_inf(sign);                         # true if +inf, or -inf (default is '+')

$x->bcmp($y);                             # compare numbers (undef,<0,=0,>0)
$x->bacmp($y);                            # compare absolutely (undef,<0,=0,>0)
$x->sign();                               # return the sign, either +, - or NaN
$x->digit($n);                           # return the nth digit, counting from right
$x->digit(-$n);                          # return the nth digit, counting from left

# The following all modify their first argument:

# set
$x->bzero();                             # set $i to 0
$x->bnan();                              # set $i to NaN
$x->bone();                               # set $x to +1
$x->bone('-');                            # set $x to -1
$x->binf();                              # set $x to inf
$x->binf('-');                           # set $x to -inf

$x->bneg();                               # negation
$x->babs();                               # absolute value
$x->bnorm();                              # normalize (no-op)
$x->bnot();                               # two's complement (bit wise not)
$x->binc();                               # increment x by 1
$x->bdec();                               # decrement x by 1

$x->badd($y);                             # addition (add $y to $x)
$x->bsub($y);                             # subtraction (subtract $y from $x)
$x->bmul($y);                             # multiplication (multiply $x by $y)
$x->bdiv($y);                             # divide, set $i to quotient
                                           # return (quo,rem) or quo if scalar

$x->bmod($y);                             # modulus
$x->bpow($y);                             # power of arguments (a**b)
$x->blsft($y);                            # left shift
```

```

$x->brsft($y);      # right shift
                    # return (quo,rem) or quo if scalar

$x->blog($base);    # logarithm of $x, base defaults to e
                    # (other bases than e not supported yet)

$x->band($y);        # bit-wise and
$x->bior($y);        # bit-wise inclusive or
$x->bxor($y);        # bit-wise exclusive or
$x->bnot();           # bit-wise not (two's complement)

$x->bsqrt();         # calculate square-root
$x->bfac();           # factorial of $x (1*2*3*4*...*x)

$x->bround($N);      # accuracy: preserver $N digits
$x->bfround($N);     # precision: round to the $Nth digit

# The following do not modify their arguments:
bgcd(@values);      # greatest common divisor
blcm(@values);      # lowest common multiplier

$x->bstr();           # return string
$x->bsstr();          # return string in scientific notation

$x->bfloor();         # return integer less or equal than $x
$x->bceil();          # return integer greater or equal than $x

$x->exponent();       # return exponent as BigInt
$x->mantissa();       # return mantissa as BigInt
$x->parts();          # return (mantissa,exponent) as BigInt

$x->length();         # number of digits (w/o sign and '.')
($l,$f) = $x->length(); # number of digits, and length of fraction

$x->precision();      # return P of $x (or global, if P of $x undef)
$x->precision($n);    # set P of $x to $n
$x->accuracy();       # return A of $x (or global, if A of $x undef)
$x->accuracy($n);     # set A $x to $n

Math::BigFloat->precision(); # get/set global P for all BigFloat objects
Math::BigFloat->accuracy();  # get/set global A for all BigFloat objects

```

DESCRIPTION

All operators (including basic math operations) are overloaded if you declare your big floating point numbers as

```
$i = new Math::BigFloat '12_3.456_789_123_456_789E-2';
```

Operations with overloaded operators preserve the arguments, which is exactly what you expect.

Canonical notation

Input to these routines are either BigFloat objects, or strings of the following four forms:

- `/^[+-]\d+$/`
- `/^[+-]\d+\.\d*$/`
- `/^[+-]\d+E[+-]?\d+$/`
- `/^[+-]\d*\.\d+E[+-]?\d+$/`

all with optional leading and trailing zeros and/or spaces. Additionally, numbers are allowed to have an underscore between any two digits.

Empty strings as well as other illegal numbers results in 'NaN'.

`bnorm()` on a BigFloat object is now effectively a no-op, since the numbers are always stored in normalized form. On a string, it creates a BigFloat object.

Output

Output values are BigFloat objects (normalized), except for `bstr()` and `bsstr()`.

The string output will always have leading and trailing zeros stripped and drop a plus sign. `bstr()` will give you always the form with a decimal point, while `bsstr()` (for scientific) gives you the scientific notation.

Input	<code>bstr()</code>	<code>bsstr()</code>
'-0'	'0'	'0E1'
' -123 123 123 '	'-123123123'	'-123123123E0'
'00.0123'	'0.0123'	'123E-4'
'123.45E-2'	'1.2345'	'12345E-4'
'10E+3'	'10000'	'1E4'

Some routines (`is_odd()`, `is_even()`, `is_zero()`, `is_one()`, `is_nan()`) return true or false, while others (`bcmp()`, `bacmp()`) return either undef, <0, 0 or 0 and are suited for sort.

Actual math is done by using BigInts to represent the mantissa and exponent. The sign `/^[+-]$/` is stored separately. The string 'NaN' is used to represent the result when input arguments are not numbers, as well as the result of dividing by zero.

`mantissa()`, `exponent()` and `parts()`

`mantissa()` and `exponent()` return the said parts of the BigFloat as BigInts such that:

```
$m = $x->mantissa();
$e = $x->exponent();
$y = $m * ( 10 ** $e );
print "ok\n" if $x == $y;
```

`< ($m,$e) = $x->parts();` is just a shortcut giving you both of them.

A zero is represented and returned as 0E1, **not** 0E0 (after Knuth).

Currently the mantissa is reduced as much as possible, favouring higher exponents over lower ones (e.g. returning 1e7 instead of 10e6 or 10000000e0). This might change in the future, so do not depend on it.

Accuracy vs. Precision

See also: [Rounding/Rounding](#).

Math::BigFloat supports both precision and accuracy. For a full documentation, examples and tips on these topics please see the large section in [Math::BigInt](#).

Since things like `sqrt(2)` or `1/3` must be presented with a limited precision lest an operation consumes all resources, each operation produces no more than `Math::BigFloat::precision()` digits.

In case the result of one operation has more precision than specified, it is rounded. The rounding mode taken is either the default mode, or the one supplied to the operation after the *scale*:

```
$x = Math::BigFloat->new(2);
Math::BigFloat::precision(5);           # 5 digits max
$y = $x->copy()->bdiv(3);                 # will give 0.66666
$y = $x->copy()->bdiv(3,6);               # will give 0.666666
$y = $x->copy()->bdiv(3,6,'odd');         # will give 0.666667
Math::BigFloat::round_mode('zero');
$y = $x->copy()->bdiv(3,6);               # will give 0.666666
```

Rounding

`ffround (+$scale)`

Rounds to the `$scale`'th place left from the '.', counting from the dot. The first digit is numbered 1.

`ffround (-$scale)`

Rounds to the `$scale`'th place right from the '.', counting from the dot.

`ffround (0)`

Rounds to an integer.

`fround (+$scale)`

Preserves accuracy to `$scale` digits from the left (aka significant digits) and pads the rest with zeros. If the number is between 1 and -1, the significant digits count from the first non-zero after the '.'.

`fround (-$scale)` and `fround (0)`

These are effectively no-ops.

All rounding functions take as a second parameter a rounding mode from one of the following: 'even', 'odd', '+inf', '-inf', 'zero' or 'trunc'.

The default rounding mode is 'even'. By using <

`Math::BigFloat::round_mode($round_mode);` you can get and set the default mode for subsequent rounding. The usage of `Math::BigFloat::$round_mode` is no longer supported. The second parameter to the round functions then overrides the default temporarily.

The `< as_number()` function returns a `BigInt` from a `Math::BigFloat`. It uses 'trunc' as rounding mode to make it equivalent to:

```
$x = 2.5;
$y = int($x) + 2;
```

You can override this by passing the desired rounding mode as parameter to `as_number()`:

```
$x = Math::BigFloat->new(2.5);
$y = $x->as_number('odd');      # $y = 3
```

EXAMPLES

```
# not ready yet
```

Autocreating constants

After use `Math::BigFloat ':constant'` all the floating point constants in the given scope are converted to `Math::BigFloat`. This conversion happens at compile time.

In particular

```
perl -MMath::BigFloat=:constant -e 'print 2E-100, "\n"'
```

prints the value of 2E-100. Note that without conversion of constants the expression 2E-100 will be calculated as normal floating point number.

Please note that ':constant' does not affect integer constants, nor binary nor hexadecimal constants. Use [bignum](#) or [Math::BigInt](#) to get this to work.

Math library

Math with the numbers is done (by default) by a module called `Math::BigInt::Calc`. This is equivalent to saying:

```
use Math::BigFloat lib => 'Calc';
```

You can change this by using:

```
use Math::BigFloat lib => 'BitVect';
```

The following would first try to find `Math::BigInt::Foo`, then `Math::BigInt::Bar`, and when this also fails, revert to `Math::BigInt::Calc`:

```
use Math::BigFloat lib => 'Foo,Math::BigInt::Bar';
```

`Calc.pm` uses as internal format an array of elements of some decimal base (usually `1e7`, but this might be different for some systems) with the least significant digit first, while `BitVect.pm` uses a bit vector of base 2, most significant bit first. Other modules might use even different means of representing the numbers. See the respective module documentation for further details.

Please note that `Math::BigFloat` does **not** use the denoted library itself, but it merely passes the `lib` argument to `Math::BigInt`. So, instead of the need to do:

```
use Math::BigInt lib => 'GMP';
use Math::BigFloat;
```

you can roll it all into one line:

```
use Math::BigFloat lib => 'GMP';
```

Use the `lib`, Luke! And see [Using Math::BigInt::Lite](#) for more details.

Using Math::BigInt::Lite

It is possible to use [Math::BigInt::Lite](#) with `Math::BigFloat`:

```
# 1
use Math::BigFloat with => 'Math::BigInt::Lite';
```

There is no need to "use `Math::BigInt`" or "use `Math::BigInt::Lite`", but you can combine these if you want. For instance, you may want to use `Math::BigInt` objects in your main script, too.

```
# 2
use Math::BigInt;
use Math::BigFloat with => 'Math::BigInt::Lite';
```

Of course, you can combine this with the `lib` parameter.

```
# 3
use Math::BigFloat with => 'Math::BigInt::Lite', lib => 'GMP,Pari';
```

If you want to use `Math::BigInt`'s, too, simple add a `Math::BigInt` **before**:

```
# 4
use Math::BigInt;
use Math::BigFloat with => 'Math::BigInt::Lite', lib => 'GMP,Pari';
```

Notice that the module with the last `lib` will "win" and thus its `lib` will be used if the `lib` is available:

```
# 5
use Math::BigInt lib => 'Bar,Baz';
use Math::BigFloat with => 'Math::BigInt::Lite', lib => 'Foo';
```

That would try to load `Foo`, `Bar`, `Baz` and `Calc` (in that order). Or in other words, `Math::BigFloat` will try to retain previously loaded `libs` when you don't specify it one.

Actually, the `lib` loading order would be "`Bar,Baz,Calc`", and then "`Foo,Bar,Baz,Calc`", but independent of which `lib` exists, the result is the same as trying the latter load alone, except for the fact that `Bar` or `Baz` might be loaded needlessly in an intermediate step

The old way still works though:

```
# 6
use Math::BigInt lib => 'Bar,Baz';
```

```
use Math::BigFloat;
```

But **examples #3 and #4 are recommended** for usage.

BUGS

- The following does not work yet:

```
$m = $x->mantissa();
$e = $x->exponent();
$y = $m * ( 10 ** $e );
print "ok\n" if $x == $y;
```

- There is no `fmod()` function yet.

CAVEAT

`stringify`, `bstr()`

Both `stringify` and `bstr()` now drop the leading '+'. The old code would return '+1.23', the new returns '1.23'. See the documentation in [Math::BigInt](#) for reasoning and details.

`bdiv`

The following will probably not do what you expect:

```
print $c->bdiv(123.456), "\n";
```

It prints both quotient and remainder since `print` works in list context. Also, `bdiv()` will modify `$c`, so be carefull. You probably want to use

```
print $c / 123.456, "\n";
print scalar $c->bdiv(123.456), "\n"; # or if you want to modify $c
```

instead.

Modifying and =

Beware of:

```
$x = Math::BigFloat->new(5);
$y = $x;
```

It will not do what you think, e.g. making a copy of `$x`. Instead it just makes a second reference to the **same** object and stores it in `$y`. Thus anything that modifies `$x` will modify `$y`, and vice versa.

```
$x->bmul(2);
print "$x, $y\n";          # prints '10, 10'
```

If you want a true copy of `$x`, use:

```
$y = $x->copy();
```

See also the documentation in [overload](#) regarding `=`.

`bpow`

`bpow()` now modifies the first argument, unlike the old code which left it alone and only returned the result. This is to be consistent with `badd()` etc. The first will modify `$x`, the second one won't:

```
print bpow($x, $i), "\n";          # modify $x
print $x->bpow($i), "\n";          # ditto
print $x ** $i, "\n";              # leave $x alone
```

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

AUTHORS

Mark Biggar, overloaded interface by Ilya Zakharevich. Completely rewritten by Tels <http://bloodgate.com> in 2001.

NAME

Math::BigInt – Arbitrary size integer math package

SYNOPSIS

```
use Math::BigInt;

# Number creation
$x = Math::BigInt->new($str);           # defaults to 0
$nan = Math::BigInt->bnan();             # create a NotANumber
$zero = Math::BigInt->bzero();           # create a +0
$inf = Math::BigInt->binf();             # create a +inf
$inf = Math::BigInt->binf('-');          # create a -inf
$one = Math::BigInt->bone();             # create a +1
$one = Math::BigInt->bone('-');          # create a -1

# Testing
$x->is_zero();                          # true if arg is +0
$x->is_nan();                           # true if arg is NaN
$x->is_one();                           # true if arg is +1
$x->is_one('-');                        # true if arg is -1
$x->is_odd();                           # true if odd, false for even
$x->is_even();                          # true if even, false for odd
$x->is_positive();                      # true if >= 0
$x->is_negative();                     # true if < 0
$x->is_inf(sign);                      # true if +inf, or -inf (sign is default '+')
$x->is_int();                           # true if $x is an integer (not a float)

$x->bcmp($y);                           # compare numbers (undef,<0,=0,>0)
$x->bacmp($y);                          # compare absolutely (undef,<0,=0,>0)
$x->sign();                             # return the sign, either +, - or NaN
$x->digit($n);                          # return the nth digit, counting from right
$x->digit(-$n);                         # return the nth digit, counting from left

# The following all modify their first argument:

# set
$x->bzero();                             # set $x to 0
$x->bnan();                              # set $x to NaN
$x->bone();                               # set $x to +1
$x->bone('-');                           # set $x to -1
$x->binf();                              # set $x to inf
$x->binf('-');                           # set $x to -inf

$x->bneg();                              # negation
$x->babs();                              # absolute value
$x->bnorm();                             # normalize (no-op)
$x->bnot();                              # two's complement (bit wise not)
$x->binc();                              # increment x by 1
$x->bdec();                              # decrement x by 1

$x->badd($y);                            # addition (add $y to $x)
$x->bsub($y);                            # subtraction (subtract $y from $x)
$x->bmul($y);                            # multiplication (multiply $x by $y)
$x->bdiv($y);                            # divide, set $x to quotient
                                           # return (quo,rem) or quo if scalar

$x->bmod($y);                            # modulus (x % y)
$x->bmodpow($exp,$mod);                  # modular exponentiation (($num**$exp) % $mod)
```

```

$x->bmodinv($mod);          # the inverse of $x in the given modulus $mod
$x->bpow($y);               # power of arguments (x ** y)
$x->blsft($y);              # left shift
$x->brsft($y);              # right shift
$x->blsft($y,$n);           # left shift, by base $n (like 10)
$x->brsft($y,$n);           # right shift, by base $n (like 10)

$x->band($y);               # bitwise and
$x->bior($y);               # bitwise inclusive or
$x->bxor($y);               # bitwise exclusive or
$x->bnot();                 # bitwise not (two's complement)

$x->bsqrt();                # calculate square-root
$x->bfac();                 # factorial of $x (1*2*3*4*...*x)

$x->round($A,$P,$round_mode); # round to accuracy or precision using mode $r
$x->bround($N);              # accuracy: preserve $N digits
$x->bfround($N);             # round to $Nth digit, no-op for BigInts

# The following do not modify their arguments in BigInt, but do in BigFloat:
$x->bfloor();                # return integer less or equal than $x
$x->bceil();                 # return integer greater or equal than $x

# The following do not modify their arguments:
bgcd(@values);              # greatest common divisor (no OO style)
blcm(@values);              # lowest common multiplier (no OO style)

$x->length();               # return number of digits in number
($x,$f) = $x->length();     # length of number and length of fraction part,
                           # latter is always 0 digits long for BigInt's

$x->exponent();              # return exponent as BigInt
$x->mantissa();              # return (signed) mantissa as BigInt
$x->parts();                 # return (mantissa,exponent) as BigInt
$x->copy();                  # make a true copy of $x (unlike $y = $x;)
$x->as_number();             # return as BigInt (in BigInt: same as copy())

# conversation to string
$x->bstr();                  # normalized string
$x->bsstr();                 # normalized string in scientific notation
$x->as_hex();                # as signed hexadecimal string with prefixed 0x
$x->as_bin();                # as signed binary string with prefixed 0b

Math::BigInt->config();      # return hash containing configuration/version

# precision and accuracy (see section about rounding for more)
$x->precision();             # return P of $x (or global, if P of $x undef)
$x->precision($n);          # set P of $x to $n
$x->accuracy();              # return A of $x (or global, if A of $x undef)
$x->accuracy($n);           # set A $x to $n

Math::BigInt->precision();   # get/set global P for all BigInt objects
Math::BigInt->accuracy();    # get/set global A for all BigInt objects

```

DESCRIPTION

All operators (including basic math operations) are overloaded if you declare your big integers as

```
$i = new Math::BigInt '123_456_789_123_456_789';
```

Operations with overloaded operators preserve the arguments which is exactly what you expect.

Canonical notation

Big integer values are strings of the form `/^[+-]\d+$/` with leading zeros suppressed.

<code>'-0'</code>	canonical value <code>'-0'</code> , normalized <code>'0'</code>
<code>'-123_123_123'</code>	canonical value <code>'-123123123'</code>
<code>'1_23_456_7890'</code>	canonical value <code>'1234567890'</code>

Input

Input values to these routines may be either `Math::BigInt` objects or strings of the form `/^[+-]?[\d]+\.[\d]*E?[+-]?[\d]*$/`.

You can include one underscore between any two digits. The input string may have leading and trailing whitespace, which will be ignored. In later versions, a more strict (no whitespace at all) or more lax (whitespace allowed everywhere) input checking will also be possible.

This means integer values like 1.01E2 or even 1000E-2 are also accepted. Non integer values result in NaN.

`Math::BigInt::new()` defaults to 0, while `Math::BigInt::new('')` results in 'NaN'.

`bnorm()` on a `BigInt` object is now effectively a no-op, since the numbers are always stored in normalized form. On a string, it creates a `BigInt` object.

Output

Output values are `BigInt` objects (normalized), except for `bstr()`, which returns a string in normalized form. Some routines (`is_odd()`, `is_even()`, `is_zero()`, `is_one()`, `is_nan()`) return true or false, while others (`bcmp()`, `bacmp()`) return either undef, <0, 0 or 0 and are suited for sort.

METHODS

Each of the methods below accepts three additional parameters. These arguments `$A`, `$P` and `$R` are accuracy, precision and round_mode. Please see more in the section about [ACCURACY](#) and [ROUNDIND](#).

config

```
use Data::Dumper;

print Dumper ( Math::BigInt->config() );
```

Returns a hash containing the configuration, e.g. the version number, lib loaded etc.

accuracy

```
$x->accuracy(5);           # local for $x
$class->accuracy(5);       # global for all members of $class
```

Set or get the global or local accuracy, aka how many significant digits the results have. Please see the section about [ACCURACY AND PRECISION](#) for further details.

Value must be greater than zero. Pass an undef value to disable it:

```
$x->accuracy(undef);
Math::BigInt->accuracy(undef);
```

Returns the current accuracy. For `$x->accuracy()` it will return either the local accuracy, or if not defined, the global. This means the return value represents the accuracy that will be in effect for `$x`:

```
$y = Math::BigInt->new(1234567);           # unrounded
print Math::BigInt->accuracy(4), "\n";      # set 4, print 4
$x = Math::BigInt->new(123456);             # will be automatically rounded
print "$x $y\n";                           # '123500 1234567'
print $x->accuracy(), "\n";                 # will be 4
print $y->accuracy(), "\n";                 # also 4, since global is 4
print Math::BigInt->accuracy(5), "\n";      # set to 5, print 5
```



```
print $x->accuracy(), "\n";      # still 4
print $y->accuracy(), "\n";      # 5, since global is 5
```

brsft

```
$x->brsft($y, $n);
```

Shifts x right by y in base n . Default is base 2, used are usually 10 and 2, but others work, too.

Right shifting usually amounts to dividing x by $n^{**}y$ and truncating the result:

```
$x = Math::BigInt->new(10);
$x->brsft(1);                      # same as $x >> 1: 5
$x = Math::BigInt->new(1234);
$x->brsft(2, 10);                  # result 12
```

There is one exception, and that is base 2 with negative x :

```
$x = Math::BigInt->new(-5);
print $x->brsft(1);
```

This will print -3 , not -2 (as it would if you divide -5 by 2 and truncate the result).

new

```
$x = Math::BigInt->new($str, $A, $P, $R);
```

Creates a new BigInt object from a string or another BigInt object. The input is accepted as decimal, hex (with leading '0x') or binary (with leading '0b').

bnan

```
$x = Math::BigInt->bnan();
```

Creates a new BigInt object representing NaN (Not A Number). If used on an object, it will set it to NaN:

```
$x->bnan();
```

bzero

```
$x = Math::BigInt->bzero();
```

Creates a new BigInt object representing zero. If used on an object, it will set it to zero:

```
$x->bzero();
```

binf

```
$x = Math::BigInt->binf($sign);
```

Creates a new BigInt object representing infinity. The optional argument is either '-' or '+', indicating whether you want infinity or minus infinity. If used on an object, it will set it to infinity:

```
$x->binf();
$x->binf('-');
```

bone

```
$x = Math::BigInt->binf($sign);
```

Creates a new BigInt object representing one. The optional argument is either '-' or '+', indicating whether you want one or minus one. If used on an object, it will set it to one:

```
$x->bone();                      # +1
$x->bone('-');                   # -1
```

is_one()/is_zero()/is_nan()/is_inf()

```
$x->is_zero();                   # true if arg is +0
$x->is_nan();                    # true if arg is NaN
$x->is_one();                    # true if arg is +1
```

```

$x->is_one('-');          # true if arg is -1
$x->is_inf();             # true if +inf
$x->is_inf('-');          # true if -inf (sign is default '+')

```

These methods all test the BigInt for being one specific value and return true or false depending on the input. These are faster than doing something like:

```
if ($x == 0)
```

is_positive()/is_negative()

```

$x->is_positive();        # true if >= 0
$x->is_negative();        # true if < 0

```

The methods return true if the argument is positive or negative, respectively. NaN is neither positive nor negative, while +inf counts as positive, and -inf is negative. A zero is positive.

These methods are only testing the sign, and not the value.

is_odd()/is_even()/is_int()

```

$x->is_odd();             # true if odd, false for even
$x->is_even();            # true if even, false for odd
$x->is_int();             # true if $x is an integer

```

The return true when the argument satisfies the condition. NaN, +inf, -inf are not integers and are neither odd nor even.

bcmp

```
$x->bcmp($y);
```

Compares \$x with \$y and takes the sign into account. Returns -1, 0, 1 or undef.

bacmp

```
$x->bacmp($y);
```

Compares \$x with \$y while ignoring their. Returns -1, 0, 1 or undef.

sign

```
$x->sign();
```

Return the sign, of \$x, meaning either +, -, -inf, +inf or NaN.

bcmp

```
$x->digit($n);           # return the nth digit, counting from right
```

bneg

```
$x->bneg();
```

Negate the number, e.g. change the sign between '+' and '-', or between '+inf' and '-inf', respectively. Does nothing for NaN or zero.

babs

```
$x->babs();
```

Set the number to it's absolute value, e.g. change the sign from '-' to '+' and from '-inf' to '+inf', respectively. Does nothing for NaN or positive numbers.

bnorm

```
$x->bnorm();             # normalize (no-op)
```

bnot

```
$x->bnot();              # two's complement (bit wise not)
```

binc

```
$x->binc();           # increment x by 1
```

bdec

```
$x->bdec();           # decrement x by 1
```

badd

```
$x->badd($y);          # addition (add $y to $x)
```

```
$x->bsub($y);          # subtraction (subtract $y from $x)
```

bmul

```
$x->bmul($y);          # multiplication (multiply $x by $y)
```

bdiv

```
$x->bdiv($y);          # divide, set $x to quotient
                        # return (quo,rem) or quo if scalar
```

bmod

```
$x->bmod($y);           # modulus (x % y)
```

bmodinv

```
$num->bmodinv($mod);    # modular inverse
```

Returns the inverse of \$num in the given modulus \$mod. ‘NaN’ is returned unless \$num is relatively prime to \$mod, i.e. unless `bgcd($num, $mod) == 1`.

bmodpow

```
$num->bmodpow($exp, $mod); # modular exponentation ($num**$exp % $mod)
```

Returns the value of \$num taken to the power \$exp in the modulus \$mod using binary exponentation. `bmodpow` is far superior to writing

```
$num ** $exp % $mod
```

because `bmodpow` is much faster—it reduces internal variables into the modulus whenever possible, so it operates on smaller numbers.

`bmodpow` also supports negative exponents.

```
bmodpow($num, -1, $mod)
```

is exactly equivalent to

```
bmodinv($num, $mod)
```

bpow

```
$x->bpow($y);           # power of arguments (x ** y)
```

blsft

```
$x->blsft($y);           # left shift
$x->blsft($y, $n);        # left shift, by base $n (like 10)
```

brsft

```
$x->brsft($y);           # right shift
$x->brsft($y, $n);        # right shift, by base $n (like 10)
```

band

```
$x->band($y);           # bitwise and
```

bior

```
$x->bior($y); # bitwise inclusive or
```

bxor

```
$x->bxor($y); # bitwise exclusive or
```

bnot

```
$x->bnot(); # bitwise not (two's complement)
```

bsqrt

```
$x->bsqrt(); # calculate square-root
```

bfac

```
$x->bfac(); # factorial of $x (1*2*3*4*...$x)
```

round

```
$x->round($A,$P,$round_mode); # round to accuracy or precision using mode $r
```

bround

```
$x->bround($N); # accuracy: preserve $N digits
```

bfround

```
$x->bfround($N); # round to $Nth digit, no-op for BigInts
```

bfloor

```
$x->bfloor();
```

Set `$x` to the integer less or equal than `$x`. This is a no-op in `BigInt`, but does change `$x` in `BigFloat`.

bceil

```
$x->bceil();
```

Set `$x` to the integer greater or equal than `$x`. This is a no-op in `BigInt`, but does change `$x` in `BigFloat`.

bgcd

```
bgcd(@values); # greatest common divisor (no OO style)
```

blcm

```
blcm(@values); # lowest common multiplier (no OO style)
```

head2 length

```
$x->length();
($x1,$f1) = $x->length();
```

Returns the number of digits in the decimal representation of the number. In list context, returns the length of the integer and fraction part. For `BigInt`'s, the length of the fraction part will always be 0.

exponent

```
$x->exponent();
```

Return the exponent of `$x` as `BigInt`.

mantissa

```
$x->mantissa();
```

Return the signed mantissa of `$x` as `BigInt`.

parts

```
$x->parts(); # return (mantissa,exponent) as BigInt
```

copy

```
$x->copy();
```

make a true copy of \$x (unlike \$y = \$x;)

as_number

```
$x->as_number();
```

return as BigInt (in BigInt: same as copy())

bsrt

```
$x->bstr();
```

normalized string

bsstr

```
$x->bsstr();
```

normalized string in scientific notation

as_hex

```
$x->as_hex();
```

as signed hexadecimal string with prefixed 0x

as_bin

```
$x->as_bin();
```

as signed binary string with prefixed 0b

ACCURACY and PRECISION

Since version v1.33, Math::BigInt and Math::BigFloat have full support for accuracy and precision based rounding, both automatically after every operation as well as manually.

This section describes the accuracy/precision handling in Math::Big* as it used to be and as it is now, complete with an explanation of all terms and abbreviations.

Not yet implemented things (but with correct description) are marked with ‘!’, things that need to be answered are marked with ‘?’.

In the next paragraph follows a short description of terms used here (because these may differ from terms used by others people or documentation).

During the rest of this document, the shortcuts A (for accuracy), P (for precision), F (fallback) and R (rounding mode) will be used.

Precision P

A fixed number of digits before (positive) or after (negative) the decimal point. For example, 123.45 has a precision of -2. 0 means an integer like 123 (or 120). A precision of 2 means two digits to the left of the decimal point are zero, so 123 with P = 1 becomes 120. Note that numbers with zeros before the decimal point may have different precisions, because 1200 can have p = 0, 1 or 2 (depending on what the initial value was). It could also have p < 0, when the digits after the decimal point are zero.

The string output (of floating point numbers) will be padded with zeros:

Initial value	P	A	Result	String
1234.01	-3		1000	1000
1234	-2		1200	1200
1234.5	-1		1230	1230
1234.001	1		1234	1234.0
1234.01	0		1234	1234
1234.01	2		1234.01	1234.01
1234.01	5		1234.01	1234.01000

For BigInts, no padding occurs.

Accuracy A

Number of significant digits. Leading zeros are not counted. A number may have an accuracy greater than the non-zero digits when there are zeros in it or trailing zeros. For example, 123.456 has A of 6, 10203 has 5, 123.0506 has 7, 123.450000 has 8 and 0.000123 has 3.

The string output (of floating point numbers) will be padded with zeros:

Initial value	P	A	Result	String
1234.01		3	1230	1230
1234.01		6	1234.01	1234.01
1234.1		8	1234.1	1234.1000

For BigInts, no padding occurs.

Fallback F

When both A and P are undefined, this is used as a fallback accuracy when dividing numbers.

Rounding mode R

When rounding a number, different ‘styles’ or ‘kinds’ of rounding are possible. (Note that random rounding, as in `Math::Round`, is not implemented.)

‘trunc’

truncation invariably removes all digits following the rounding place, replacing them with zeros. Thus, 987.65 rounded to tens ($P=1$) becomes 980, and rounded to the fourth sigdig becomes 987.6 ($A=4$). 123.456 rounded to the second place after the decimal point ($P=-2$) becomes 123.46.

All other implemented styles of rounding attempt to round to the "nearest digit." If the digit D immediately to the right of the rounding place (skipping the decimal point) is greater than 5, the number is incremented at the rounding place (possibly causing a cascade of incrementation): e.g. when rounding to units, 0.9 rounds to 1, and -19.9 rounds to -20 . If $D < 5$, the number is similarly truncated at the rounding place: e.g. when rounding to units, 0.4 rounds to 0, and -19.4 rounds to -19 .

However the results of other styles of rounding differ if the digit immediately to the right of the rounding place (skipping the decimal point) is 5 and if there are no digits, or no digits other than 0, after that 5. In such cases:

‘even’

rounds the digit at the rounding place to 0, 2, 4, 6, or 8 if it is not already. E.g., when rounding to the first sigdig, 0.45 becomes 0.4, -0.55 becomes -0.6 , but 0.4501 becomes 0.5.

‘odd’

rounds the digit at the rounding place to 1, 3, 5, 7, or 9 if it is not already. E.g., when rounding to the first sigdig, 0.45 becomes 0.5, -0.55 becomes -0.5 , but 0.5501 becomes 0.6.

‘+inf’

round to plus infinity, i.e. always round up. E.g., when rounding to the first sigdig, 0.45 becomes 0.5, -0.55 becomes -0.5 , and 0.4501 also becomes 0.5.

‘-inf’

round to minus infinity, i.e. always round down. E.g., when rounding to the first sigdig, 0.45 becomes 0.4, -0.55 becomes -0.6 , but 0.4501 becomes 0.5.

‘zero’

round to zero, i.e. positive numbers down, negative ones up. E.g., when rounding to the first sigdig, 0.45 becomes 0.4, -0.55 becomes -0.5 , but 0.4501 becomes 0.5.

The handling of A & P in MBI/MBF (the old core code shipped with Perl versions $\leq 5.7.2$) is like this:

Precision

- * `ffround($p)` is able to round to \$p number of digits after the decimal point
- * otherwise P is unused

Accuracy (significant digits)

- * `fround($a)` rounds to \$a significant digits
 - * only `fdiv()` and `fsqrt()` take A as (optional) parameter
 - + other operations simply create the same number (`fneg` etc), or more (`fmul`) of digits
 - + rounding/truncating is only done when explicitly calling one of `fround` or `ffround`, and never for `BigInt` (not implemented)
 - * `fsqrt()` simply hands its accuracy argument over to `fdiv`.
 - * the documentation and the comment in the code indicate two different ways on how `fdiv()` determines the maximum number of digits it should calculate, and the actual code does yet another thing
- POD:
- ```
max($Math::BigFloat::div_scale,length(dividend)+length(divisor))
```
- Comment:
- ```
result has at most max(scale, length(dividend), length(divisor)) digits
```
- Actual code:
- ```
scale = max(scale, length(dividend)-1,length(divisor)-1);
scale += length(divisor) - length(dividend);
```
- So for `lx = 3`, `ly = 9`, `scale = 10`, `scale` will actually be 16 (`10+9-3`). Actually, the 'difference' added to the scale is calculated from the number of "significant digits" in dividend and divisor, which is derived by looking at the length of the mantissa. Which is wrong, since it includes the + sign (oops) and actually gets 2 for `'+100'` and 4 for `'+101'`. Oups again. Thus `124/3` with `div_scale=1` will get you `'41.3'` based on the strange assumption that 124 has 3 significant digits, while `120/7` will get you `'17'`, not `'17.1'` since 120 is thought to have 2 significant digits. The rounding after the division then uses the remainder and `$y` to determine whether it must round up or down.
- ? I have no idea which is the right way. That's why I used a slightly more
- ? simple scheme and tweaked the few failing testcases to match it.

This is how it works now:

### Setting/Accessing

- \* You can set the A global via `Math::BigInt->accuracy()` or `Math::BigFloat->accuracy()` or whatever class you are using.
- \* You can also set P globally by using `Math::SomeClass->precision()` likewise.
- \* Globals are classwide, and not inherited by subclasses.
- \* to undefine A, use `Math::SomeClass->accuracy(undef);`
- \* to undefine P, use `Math::SomeClass->precision(undef);`
- \* Setting `Math::SomeClass->accuracy()` clears automatically `Math::SomeClass->precision()`, and vice versa.
- \* To be valid, A must be `> 0`, P can have any value.
- \* If P is negative, this means round to the P'th place to the right of the decimal point; positive values mean to the left of the decimal point. P of 0 means round to integer.
- \* to find out the current global A, take `Math::SomeClass->accuracy()`
- \* to find out the current global P, take `Math::SomeClass->precision()`
- \* use `$x->accuracy()` respective `$x->precision()` for the local setting of `$x`.
- \* Please note that `$x->accuracy()` respective `$x->precision()` fall back to the defined globals, when `$x's` A or P is not set.

### Creating numbers

- \* When you create a number, you can give it's desired A or P via:
 

```
$x = Math::BigInt->new($number,$A,$P);
```

- \* Only one of A or P can be defined, otherwise the result is NaN
- \* If no A or P is give (`$x = Math::BigInt->new($number)` form), then the globals (if set) will be used. Thus changing the global defaults later on will not change the A or P of previously created numbers (i.e., A and P of \$x will be what was in effect when \$x was created)
- \* If given undef for A and P, B<no> rounding will occur, and the globals will B<not> be used. This is used by subclasses to create numbers without suffering rounding in the parent. Thus a subclass is able to have it's own globals enforced upon creation of a number by using  
`$x = Math::BigInt->new($number,undef,undef) :`

```

 use Math::BigInt::SomeSubclass;
 use Math::BigInt;

 Math::BigInt->accuracy(2);
 Math::BigInt::SomeSubClass->accuracy(3);
 $x = Math::BigInt::SomeSubClass->new(1234);

```

\$x is now 1230, and not 1200. A subclass might choose to implement this otherwise, e.g. falling back to the parent's A and P.

### Usage

- \* If A or P are enabled/defined, they are used to round the result of each operation according to the rules below
- \* Negative P is ignored in Math::BigInt, since BigInts never have digits after the decimal point
- \* Math::BigFloat uses Math::BigInts internally, but setting A or P inside Math::BigInt as globals should not tamper with the parts of a BigFloat. Thus a flag is used to mark all Math::BigFloat numbers as 'never round'

### Precedence

- \* It only makes sense that a number has only one of A or P at a time. Since you can set/get both A and P, there is a rule that will practically enforce only A or P to be in effect at a time, even if both are set. This is called precedence.
- \* If two objects are involved in an operation, and one of them has A in effect, and the other P, this results in an error (NaN).
- \* A takes precedence over P (Hint: A comes before P). If A is defined, it is used, otherwise P is used. If neither of them is defined, nothing is used, i.e. the result will have as many digits as it can (with an exception for fdiv/fsqrt) and will not be rounded.
- \* There is another setting for fdiv() (and thus for fsqrt()). If neither of A or P is defined, fdiv() will use a fallback (F) of \$div\_scale digits. If either the dividend's or the divisor's mantissa has more digits than the value of F, the higher value will be used instead of F. This is to limit the digits (A) of the result (just consider what would happen with unlimited A and P in the case of 1/3 :-)
- \* fdiv will calculate (at least) 4 more digits than required (determined by A, P or F), and, if F is not used, round the result (this will still fail in the case of a result like 0.12345000000001 with A or P of 5, but this can not be helped - or can it?)
- \* Thus you can have the math done by on Math::Big\* class in three modes:
  - + never round (this is the default):  
 This is done by setting A and P to undef. No math operation will round the result, with fdiv() and fsqrt() as exceptions to guard against overflows. You must explicitly call bround(), bround() or round() (the latter with parameters).



Note: Once you have rounded a number, the settings will 'stick' on it and 'infect' all other numbers engaged in math operations with it, since local settings have the highest precedence. So, to get `SafeRound[tm]`, use a `copy()` before rounding like this:

```
$x = Math::BigFloat->new(12.34);
$y = Math::BigFloat->new(98.76);
$z = $x * $y; # 1218.6984
print $x->copy()->fround(3); # 12.3 (but A is now 3!)
$z = $x * $y; # still 1218.6984, without
 # copy would have been 1210!
```

+ round after each op:

After each single operation (except for testing like `is_zero()`), the method `round()` is called and the result is rounded appropriately. By setting proper values for A and P, you can have all-the-same-A or all-the-same-P modes. For example, `Math::Currency` might set A to undef, and P to -2, globally.

?Maybe an extra option that forbids local A & P settings would be in order, ?so that intermediate rounding does not 'poison' further math?

### Overriding globals

- \* you will be able to give A, P and R as an argument to all the calculation routines; the second parameter is A, the third one is P, and the fourth is R (shift right by one for binary operations like `badd`). P is used only if the first parameter (A) is undefined. These three parameters override the globals in the order detailed as follows, i.e. the first defined value wins:
  - (local: per object, global: global default, parameter: argument to sub)
  - + parameter A
  - + parameter P
  - + local A (if defined on both of the operands: smaller one is taken)
  - + local P (if defined on both of the operands: bigger one is taken)
  - + global A
  - + global P
  - + global F
- \* `fsqrt()` will hand its arguments to `fdiv()`, as it used to, only now for two arguments (A and P) instead of one

### Local settings

- \* You can set A and P locally by using `$x->accuracy()` and `$x->precision()` and thus force different A and P for different objects/numbers.
- \* Setting A or P this way immediately rounds `$x` to the new value.
- \* `$x->accuracy()` clears `$x->precision()`, and vice versa.

### Rounding

- \* the rounding routines will use the respective global or local settings. `fround()/bround()` is for accuracy rounding, while `ffround()/bfround()` is for precision
- \* the two rounding functions take as the second parameter one of the following rounding modes (R):
  - 'even', 'odd', '+inf', '-inf', 'zero', 'trunc'
- \* you can set and get the global R by using `Math::SomeClass->round_mode()` or by setting `$Math::SomeClass::round_mode`
- \* after each operation, `$result->round()` is called, and the result may eventually be rounded (that is, if A or P were set either locally,

- globally or as parameter to the operation)
- \* to manually round a number, call `$x->round($A,$P,$round_mode)`; this will round the number by using the appropriate rounding function and then normalize it.
- \* rounding modifies the local settings of the number:

```
$x = Math::BigFloat->new(123.456);
$x->accuracy(5);
$x->bround(4);
```

Here 4 takes precedence over 5, so 123.5 is the result and `$x->accuracy()` will be 4 from now on.

#### Default values

- \* R: 'even'
- \* F: 40
- \* A: undef
- \* P: undef

#### Remarks

- \* The defaults are set up so that the new code gives the same results as the old code (except in a few cases on `fdiv`):
- + Both A and P are undefined and thus will not be used for rounding after each operation.
- + `round()` is thus a no-op, unless given extra parameters A and P

## INTERNALS

The actual numbers are stored as unsigned big integers (with separate sign). You should neither care about nor depend on the internal representation; it might change without notice. Use only method calls like `< $x->sign()`; instead relying on the internal hash keys like in `< $x-{sign}`;

## MATH LIBRARY

Math with the numbers is done (by default) by a module called `Math::BigInt::Calc`. This is equivalent to saying:

```
use Math::BigInt lib => 'Calc';
```

You can change this by using:

```
use Math::BigInt lib => 'BitVect';
```

The following would first try to find `Math::BigInt::Foo`, then `Math::BigInt::Bar`, and when this also fails, revert to `Math::BigInt::Calc`:

```
use Math::BigInt lib => 'Foo,Math::BigInt::Bar';
```

`Calc.pm` uses as internal format an array of elements of some decimal base (usually 1e5 or 1e7) with the least significant digit first, while `BitVect.pm` uses a bit vector of base 2, most significant bit first. Other modules might use even different means of representing the numbers. See the respective module documentation for further details.

## SIGN

The sign is either '+', '-', 'NaN', '+inf' or '-inf' and stored separately.

A sign of 'NaN' is used to represent the result when input arguments are not numbers or as a result of 0/0. '+inf' and '-inf' represent plus respectively minus infinity. You will get '+inf' when dividing a positive number by 0, and '-inf' when dividing any negative number by 0.

**mantissa(), exponent() and parts()**

`mantissa()` and `exponent()` return the said parts of the `BigInt` such that:

```
$m = $x->mantissa();
$e = $x->exponent();
$y = $m * (10 ** $e);
print "ok\n" if $x == $y;
```

`< ($m,$e) = $x->parts()` is just a shortcut that gives you both of them in one go. Both the returned mantissa and exponent have a sign.

Currently, for `BigInts` `$e` will be always 0, except for NaN, +inf and -inf, where it will be NaN; and for `$x == 0`, where it will be 1 (to be compatible with `Math::BigFloat`'s internal representation of a zero as 0E1).

`$m` will always be a copy of the original number. The relation between `$e` and `$m` might change in the future, but will always be equivalent in a numerical sense, e.g. `$m` might get minimized.

**EXAMPLES**

```
use Math::BigInt;

sub bint { Math::BigInt->new(shift); }

$x = Math::BigInt->bstr("1234") # string "1234"
$x = "$x"; # same as bstr()
$x = Math::BigInt->bneg("1234"); # BigInt "-1234"
$x = Math::BigInt->babs("-12345"); # BigInt "12345"
$x = Math::BigInt->bnorm("-0 00"); # BigInt "0"
$x = bint(1) + bint(2); # BigInt "3"
$x = bint(1) + "2"; # ditto (auto-BigIntify of "2")
$x = bint(1); # BigInt "1"
$x = $x + 5 / 2; # BigInt "3"
$x = $x ** 3; # BigInt "27"
$x *= 2; # BigInt "54"
$x = Math::BigInt->new(0); # BigInt "0"
$x--; # BigInt "-1"
$x = Math::BigInt->badd(4,5) # BigInt "9"
print $x->bsstr(); # 9e+0
```

Examples for rounding:

```
use Math::BigFloat;
use Test;

$x = Math::BigFloat->new(123.4567);
$y = Math::BigFloat->new(123.456789);
Math::BigFloat->accuracy(4); # no more A than 4

ok ($x->copy()->fround(),123.4); # even rounding
print $x->copy()->fround(),"\n"; # 123.4
Math::BigFloat->round_mode('odd'); # round to odd
print $x->copy()->fround(),"\n"; # 123.5
Math::BigFloat->accuracy(5); # no more A than 5
Math::BigFloat->round_mode('odd'); # round to odd
print $x->copy()->fround(),"\n"; # 123.46
$y = $x->copy()->fround(4)," \n"; # A = 4: 123.4
print "$y, ", $y->accuracy()," \n"; # 123.4, 4

Math::BigFloat->accuracy(undef); # A not important now
Math::BigFloat->precision(2); # P important
print $x->copy()->bnorm(),"\n"; # 123.46
```

```
print $x->copy()->fround(),"\n#;123.46
```

Examples for converting:

```
my $x = Math::BigInt->new('0b1'.'01' x 123);
print "bin: ",$x->as_bin()," hex: ",$x->as_hex()," dec: ",$x,"\n";
```

### Autocreating constants

After use `Math::BigInt ':constant'` all the **integer** decimal, hexadecimal and binary constants in the given scope are converted to `Math::BigInt`. This conversion happens at compile time.

In particular,

```
perl -MMath::BigInt=:constant -e 'print 2**100,"\n"'
```

prints the integer value of `2**100`. Note that without conversion of constants the expression `2**100` will be calculated as perl scalar.

Please note that strings and floating point constants are not affected, so that

```
use Math::BigInt qw/:constant/;

$x = 1234567890123456789012345678901234567890
 + 123456789123456789;
$y = '1234567890123456789012345678901234567890'
 + '123456789123456789';
```

do not work. You need an explicit `Math::BigInt-new()` around one of the operands. You should also quote large constants to protect loss of precision:

```
use Math::BigInt;

$x = Math::BigInt->new('1234567889123456789123456789123456789');
```

Without the quotes Perl would convert the large number to a floating point constant at compile time and then hand the result to `BigInt`, which results in an truncated result or a NaN.

This also applies to integers that look like floating point constants:

```
use Math::BigInt ':constant';

print ref(123e2),"\n";
print ref(123.2e2),"\n";
```

will print nothing but newlines. Use either *bignum* or *Math::BigFloat* to get this to work.

### PERFORMANCE

Using the form `$x += $y`; etc over `$x = $x + $y` is faster, since a copy of `$x` must be made in the second case. For long numbers, the copy can eat up to 20% of the work (in the case of addition/subtraction, less for multiplication/division). If `$y` is very small compared to `$x`, the form `$x += $y` is MUCH faster than `$x = $x + $y` since making the copy of `$x` takes more time then the actual addition.

With a technique called copy-on-write, the cost of copying with overload could be minimized or even completely avoided. A test implementation of COW did show performance gains for overloaded math, but introduced a performance loss due to a constant overhead for all other operations.

The rewritten version of this module is slower on certain operations, like `new()`, `bstr()` and `numify()`. The reason are that it does now more work and handles more cases. The time spent in these operations is usually gained in the other operations so that programs on the average should get faster. If they don't, please contact the author.

Some operations may be slower for small numbers, but are significantly faster for big numbers. Other operations are now constant (O(1), like `bneg()`, `babs()` etc), instead of O(N) and thus nearly always take much less time. These optimizations were done on purpose.

If you find the Calc module to slow, try to install any of the replacement modules and see if they help you.

### Alternative math libraries

You can use an alternative library to drive Math::BigInt via:

```
use Math::BigInt lib => 'Module';
```

See [MATH LIBRARY](#) for more information.

For more benchmark results see <http://bloodgate.com/perl/benchmarks.html>.

## SUBCLASSING

### Subclassing Math::BigInt

The basic design of Math::BigInt allows simple subclasses with very little work, as long as a few simple rules are followed:

- The public API must remain consistent, i.e. if a sub-class is overloading addition, the sub-class must use the same name, in this case `badd()`. The reason for this is that Math::BigInt is optimized to call the object methods directly.
- The private object hash keys like `$x-{sign}` may not be changed, but additional keys can be added, like `$x-{_custom}`.
- Accessor functions are available for all existing object hash keys and should be used instead of directly accessing the internal hash keys. The reason for this is that Math::BigInt itself has a pluggable interface which permits it to support different storage methods.

More complex sub-classes may have to replicate more of the logic internal of Math::BigInt if they need to change more basic behaviors. A subclass that needs to merely change the output only needs to overload `bstr()`.

All other object methods and overloaded functions can be directly inherited from the parent class.

At the very minimum, any subclass will need to provide it's own `new()` and can store additional hash keys in the object. There are also some package globals that must be defined, e.g.:

```
Globals
$accuracy = undef;
$precision = -2; # round to 2 decimal places
$round_mode = 'even';
$div_scale = 40;
```

Additionally, you might want to provide the following two globals to allow auto-upgrading and auto-downgrading to work correctly:

```
$upgrade = undef;
$downgrade = undef;
```

This allows Math::BigInt to correctly retrieve package globals from the subclass, like `$SubClass::precision`. See `t/Math/BigInt/Subclass.pm` or `t/Math/BigFloat/SubClass.pm` completely functional subclass examples.

Don't forget to

```
use overload;
```

in your subclass to automatically inherit the overloading from the parent. If you like, you can change part of the overloading, look at Math::String for an example.

## UPGRADING

When used like this:

```
use Math::BigInt upgrade => 'Foo::Bar';
```

certain operations will ‘upgrade’ their calculation and thus the result to the class `Foo::Bar`. Usually this is used in conjunction with `Math::BigFloat`:

```
use Math::BigInt upgrade => 'Math::BigFloat';
```

As a shortcut, you can use the module `bignum`:

```
use bignum;
```

Also good for oneliners:

```
perl -mbignum -le 'print 2 ** 255'
```

This makes it possible to mix arguments of different classes (as in  $2.5 + 2$ ) as well as preserve accuracy (as in `sqrt(3)`).

Beware: This feature is not fully implemented yet.

### Auto-upgrade

The following methods upgrade themselves unconditionally; that is if upgrade is in effect, they will always hand up their work:

```
bsqrt()
div()
blog()
```

Beware: This list is not complete.

All other methods upgrade themselves only when one (or all) of their arguments are of the class mentioned in `$upgrade` (This might change in later versions to a more sophisticated scheme):

### BUGS

#### Out of Memory!

Under Perl prior to 5.6.0 having an `use Math::BigInt ':constant';` and `eval()` in your code will crash with "Out of memory". This is probably an overload/exporter bug. You can workaround by not having `eval()` and `:constant` at the same time or upgrade your Perl to a newer version.

#### Fails to load Calc on Perl prior 5.6.0

Since `eval('use ...')` can not be used in conjunction with `:constant`, `BigInt` will fall back to `eval { require ... }` when loading the math lib on Perls prior to 5.6.0. This simple replaces `::` with `/` and thus might fail on filesystems using a different separator.

### CAVEATS

Some things might not work as you expect them. Below is documented what is known to be troublesome:

#### `stringify`, `bstr()`, `bsstr()` and `'cmp'`

Both `stringify` and `bstr()` now drop the leading `+`. The old code would return `+3`, the new returns `3`. This is to be consistent with Perl and to make `cmp` (especially with overloading) to work as you expect. It also solves problems with `Test.pm`, it's `ok()` uses `'eq'` internally.

Mark said, when asked about to drop the `+` altogether, or make only `cmp` work:

```
I agree (with the first alternative), don't add the '+' on positive
numbers. It's not as important anymore with the new internal
form for numbers. It made doing things like abs and neg easier,
but those have to be done differently now anyway.
```

So, the following examples will now work all as expected:

```
use Test;
BEGIN { plan tests => 1 }
use Math::BigInt;
```

```

my $x = new Math::BigInt 3*3;
my $y = new Math::BigInt 3*3;

ok ($x,3*3);
print "$x eq 9" if $x eq $y;
print "$x eq 9" if $x eq '9';
print "$x eq 9" if $x eq 3*3;

```

Additionally, the following still works:

```

print "$x == 9" if $x == $y;
print "$x == 9" if $x == 9;
print "$x == 9" if $x == 3*3;

```

There is now a `bsstr()` method to get the string in scientific notation aka `1e+2` instead of `100`. Be advised that overloaded `'eq'` always uses `bstr()` for comparison, but Perl will represent some numbers as `100` and others as `1e+308`. If in doubt, convert both arguments to `Math::BigInt` before doing `eq`:

```

use Test;
BEGIN { plan tests => 3 }
use Math::BigInt;

$x = Math::BigInt->new('1e56'); $y = 1e56;
ok ($x,$y); # will fail
ok ($x->bsstr(),$y); # okay
$y = Math::BigInt->new($y);
ok ($x,$y); # okay

```

Alternatively, simply use `<=` for comparisons, that will get it always right. There is not yet a way to get a number automatically represented as a string that matches exactly the way Perl represents it.

`int()`

`int()` will return (at least for Perl v5.7.1 and up) another `BigInt`, not a Perl scalar:

```

$x = Math::BigInt->new(123);
$y = int($x); # BigInt 123
$x = Math::BigFloat->new(123.45);
$y = int($x); # BigInt 123

```

In all Perl versions you can use `as_number()` for the same effect:

```

$x = Math::BigFloat->new(123.45);
$y = $x->as_number(); # BigInt 123

```

This also works for other subclasses, like `Math::String`.

It is yet unclear whether overloaded `int()` should return a scalar or a `BigInt`.

`length`

The following will probably not do what you expect:

```

$c = Math::BigInt->new(123);
print $c->length(), "\n"; # prints 30

```

It prints both the number of digits in the number and in the fraction part since `print` calls `length()` in list context. Use something like:

```

print scalar $c->length(), "\n"; # prints 3

```

`bdiv`

The following will probably not do what you expect:

```

print $c->bdiv(10000), "\n";

```

It prints both quotient and remainder since `print` calls `bdiv()` in list context. Also, `bdiv()` will modify `$c`, so be carefull. You probably want to use

```
print $c / 10000, "\n";
print scalar $c->bdiv(10000), "\n"; # or if you want to modify $c
```

instead.

The quotient is always the greatest integer less than or equal to the real-valued quotient of the two operands, and the remainder (when it is nonzero) always has the same sign as the second operand; so, for example,

```
1 / 4 => (0, 1)
1 / -4 => (-1, -3)
-3 / 4 => (-1, 1)
-3 / -4 => (0, -3)
-11 / 2 => (-5, 1)
11 / -2 => (-5, -1)
```

As a consequence, the behavior of the operator `%` agrees with the behavior of Perl's built-in `%` operator (as documented in the `perlop` manpage), and the equation

$$\$x == (\$x / \$y) * \$y + (\$x \% \$y)$$

holds true for any `$x` and `$y`, which justifies calling the two return values of `bdiv()` the quotient and remainder. The only exception to this rule are when `$y == 0` and `$x` is negative, then the remainder will also be negative. See below under "infinity handling" for the reasoning behind this.

Perl's 'use integer;' changes the behaviour of `%` and `/` for scalars, but will not change `BigInt`'s way to do things. This is because under 'use integer' Perl will do what the underlying C thinks is right and this is different for each system. If you need `BigInt`'s behaving exactly like Perl's 'use integer', bug the author to implement it ;)

### infinity handling

Here are some examples that explain the reasons why certain results occur while handling infinity:

The following table shows the result of the division and the remainder, so that the equation above holds true. Some "ordinary" cases are strewn in to show more clearly the reasoning:

| A / B = C,       | R  | so that C * B + R = A |
|------------------|----|-----------------------|
| =====            |    |                       |
| 5 / 8 = 0,       | 5  | 0 * 8 + 5 = 5         |
| 0 / 8 = 0,       | 0  | 0 * 8 + 0 = 0         |
| 0 / inf = 0,     | 0  | 0 * inf + 0 = 0       |
| 0 / -inf = 0,    | 0  | 0 * -inf + 0 = 0      |
| 5 / inf = 0,     | 5  | 0 * inf + 5 = 5       |
| 5 / -inf = 0,    | 5  | 0 * -inf + 5 = 5      |
| -5 / inf = 0,    | -5 | 0 * inf + -5 = -5     |
| -5 / -inf = 0,   | -5 | 0 * -inf + -5 = -5    |
| inf / 5 = inf,   | 0  | inf * 5 + 0 = inf     |
| -inf / 5 = -inf, | 0  | -inf * 5 + 0 = -inf   |
| inf / -5 = -inf, | 0  | -inf * -5 + 0 = inf   |
| -inf / -5 = inf, | 0  | inf * -5 + 0 = -inf   |
| 5 / 5 = 1,       | 0  | 1 * 5 + 0 = 5         |
| -5 / -5 = 1,     | 0  | 1 * -5 + 0 = -5       |
| inf / inf = 1,   | 0  | 1 * inf + 0 = inf     |
| -inf / -inf = 1, | 0  | 1 * -inf + 0 = -inf   |
| inf / -inf = -1, | 0  | -1 * -inf + 0 = inf   |
| -inf / inf = -1, | 0  | 1 * -inf + 0 = -inf   |
| 8 / 0 = inf,     | 8  | inf * 0 + 8 = 8       |



```

inf/ 0 = inf, inf inf * 0 + inf = inf
0/ 0 = NaN

```

These cases below violate the "remainder has the sign of the second of the two arguments", since they wouldn't match up otherwise.

```

 A / B = C, R so that C * B + R = A
=====
-inf/ 0 = -inf, -inf -inf * 0 + inf = -inf
-8/ 0 = -inf, -8 -inf * 0 + 8 = -8

```

### Modifying and =

Beware of:

```

$x = Math::BigFloat->new(5);
$y = $x;

```

It will not do what you think, e.g. making a copy of `$x`. Instead it just makes a second reference to the **same** object and stores it in `$y`. Thus anything that modifies `$x` (except overloaded operators) will modify `$y`, and vice versa. Or in other words, `=` is only safe if you modify your BigInts only via overloaded math. As soon as you use a method call it breaks:

```

$x->bmul(2);
print "$x, $y\n"; # prints '10, 10'

```

If you want a true copy of `$x`, use:

```

$y = $x->copy();

```

You can also chain the calls like this, this will make first a copy and then multiply it by 2:

```

$y = $x->copy()->bmul(2);

```

See also the documentation for `overload.pm` regarding `=`.

### bpow

`bpow()` (and the rounding functions) now modifies the first argument and returns it, unlike the old code which left it alone and only returned the result. This is to be consistent with `badd()` etc. The first three will modify `$x`, the last one won't:

```

print bpow($x,$i), "\n"; # modify $x
print $x->bpow($i), "\n"; # ditto
print $x **= $i, "\n"; # the same
print $x ** $i, "\n"; # leave $x alone

```

The form `$x **= $y` is faster than `$x = $x ** $y`, though.

### Overloading -\$x

The following:

```

$x = -$x;

```

is slower than

```

$x->bneg();

```

since `overload` calls `sub($x, 0, 1)`; instead of `neg($x)`. The first variant needs to preserve `$x` since it does not know that it later will get overwritten. This makes a copy of `$x` and takes  $O(N)$ , but `$x->bneg()` is  $O(1)$ .

With Copy-On-Write, this issue would be gone, but C-o-W is not implemented since it is slower for all other things.

### Mixing different object types

In Perl you will get a floating point value if you do one of the following:

```
$float = 5.0 + 2;
$float = 2 + 5.0;
$float = 5 / 2;
```

With overloaded math, only the first two variants will result in a BigInt:

```
use Math::BigInt;
use Math::BigFloat;

$mbf = Math::BigFloat->new(5);
$mbi2 = Math::BigInteger->new(5);
$mbi = Math::BigInteger->new(2);

what actually gets called:
$float = $mbf + $mbi; # $mbf->badd()
$float = $mbf / $mbi; # $mbf->bdiv()
$integer = $mbi + $mbf; # $mbi->badd()
$integer = $mbi2 / $mbi; # $mbi2->bdiv()
$integer = $mbi2 / $mbf; # $mbi2->bdiv()
```

This is because math with overloaded operators follows the first (dominating) operand, and the operation of that is called and returns thus the result. So, `Math::BigInt::bdiv()` will always return a `Math::BigInt`, regardless whether the result should be a `Math::BigFloat` or the second operand is one.

To get a `Math::BigFloat` you either need to call the operation manually, make sure the operands are already of the proper type or casted to that type via `Math::BigFloat->new()`:

```
$float = Math::BigFloat->new($mbi2) / $mbi; # = 2.5
```

Beware of simple "casting" the entire expression, this would only convert the already computed result:

```
$float = Math::BigFloat->new($mbi2 / $mbi); # = 2.0 thus wrong!
```

Beware also of the order of more complicated expressions like:

```
$integer = ($mbi2 + $mbi) / $mbf; # int / float => int
$integer = $mbi2 / Math::BigFloat->new($mbi); # ditto
```

If in doubt, break the expression into simpler terms, or cast all operands to the desired resulting type.

Scalar values are a bit different, since:

```
$float = 2 + $mbf;
$float = $mbf + 2;
```

will both result in the proper type due to the way the overloaded math works.

This section also applies to other overloaded math packages, like `Math::String`.

One solution to your problem might be [autoupgrading/upgrading](#).

### `bsqrt()`

`bsqrt()` works only good if the result is a big integer, e.g. the square root of 144 is 12, but from 12 the square root is 3, regardless of rounding mode.

If you want a better approximation of the square root, then use:

```
$x = Math::BigFloat->new(12);
Math::BigFloat->precision(0);
Math::BigFloat->round_mode('even');
print $x->copy->bsqrt(), "\n"; # 4
```

```
Math::BigFloat->precision(2);
print $x->bsqrt(), "\n"; # 3.46
print $x->bsqrt(3), "\n"; # 3.464

brsft()
```

For negative numbers in base see also [brsft/brsft](#).

## LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

## SEE ALSO

[Math::BigFloat](#) and [Math::Big](#) as well as [Math::BigInt::BitVect](#), [Math::BigInt::Pari](#) and [Math::BigInt::GMP](#).

The package at <http://search.cpan.org/search?mode=module&query=Math%3A%3ABigInt> contains more documentation including a full version history, testcases, empty subclass files and benchmarks.

## AUTHORS

Original code by Mark Biggar, overloaded interface by Ilya Zakharevich. Completely rewritten by Tels <http://bloodgate.com> in late 2000, 2001.

**NAME**

Math::BigInt::Calc – Pure Perl module to support Math::BigInt

**SYNOPSIS**

Provides support for big integer calculations. Not intended to be used by other modules (except Math::BigInt::Cached). Other modules which sport the same functions can also be used to support Math::BigInt, like Math::BigInt::Pari.

**DESCRIPTION**

In order to allow for multiple big integer libraries, Math::BigInt was rewritten to use library modules for core math routines. Any module which follows the same API as this can be used instead by using the following:

```
use Math::BigInt lib => 'libname';
```

'libname' is either the long name ('Math::BigInt::Pari'), or only the short version like 'Pari'.

**EXPORT**

The following functions **MUST** be defined in order to support the use by Math::BigInt:

|                             |                                                                                                                                                              |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>_new(string)</code>   | return ref to new object from ref to decimal string                                                                                                          |
| <code>_zero()</code>        | return a new object with value 0                                                                                                                             |
| <code>_one()</code>         | return a new object with value 1                                                                                                                             |
| <code>_str(obj)</code>      | return ref to a string representing the object                                                                                                               |
| <code>_num(obj)</code>      | returns a Perl integer/floating point number                                                                                                                 |
|                             | NOTE: because of Perl numeric notation defaults, the <code>_num</code> 'ified obj may lose accuracy due to machine-dependend floating point size limitations |
| <code>_add(obj,obj)</code>  | Simple addition of two objects                                                                                                                               |
| <code>_mul(obj,obj)</code>  | Multiplication of two objects                                                                                                                                |
| <code>_div(obj,obj)</code>  | Division of the 1st object by the 2nd                                                                                                                        |
|                             | In list context, returns (result,remainder).                                                                                                                 |
|                             | NOTE: this is integer math, so no fractional part will be returned.                                                                                          |
| <code>_sub(obj,obj)</code>  | Simple subtraction of 1 object from another                                                                                                                  |
|                             | a third, optional parameter indicates that the params are swapped. In this case, the first param needs to be preserved, while you can destroy the second.    |
|                             | sub (x,y,1) => return x - y and keep x intact!                                                                                                               |
| <code>_dec(obj)</code>      | decrement object by one (input is garant. to be > 0)                                                                                                         |
| <code>_inc(obj)</code>      | increment object by one                                                                                                                                      |
| <code>_acmp(obj,obj)</code> | <=> operator for objects (return -1, 0 or 1)                                                                                                                 |
| <code>_len(obj)</code>      | returns count of the decimal digits of the object                                                                                                            |
| <code>_digit(obj,n)</code>  | returns the n'th decimal digit of object                                                                                                                     |
| <code>_is_one(obj)</code>   | return true if argument is +1                                                                                                                                |
| <code>_is_zero(obj)</code>  | return true if argument is 0                                                                                                                                 |
| <code>_is_even(obj)</code>  | return true if argument is even (0,2,4,6..)                                                                                                                  |
| <code>_is_odd(obj)</code>   | return true if argument is odd (1,3,5,7..)                                                                                                                   |
| <code>_copy</code>          | return a ref to a true copy of the object                                                                                                                    |
| <code>_check(obj)</code>    | check whether internal representation is still intact                                                                                                        |
|                             | return 0 for ok, otherwise error message as string                                                                                                           |

The following functions are optional, and can be defined if the underlying lib has a fast way to do them. If undefined, Math::BigInt will use pure Perl (hence slow) fallback routines to emulate these:

|                              |                                                                                                                                                |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>_from_hex(str)</code>  | return ref to new object from ref to hexadecimal string                                                                                        |
| <code>_from_bin(str)</code>  | return ref to new object from ref to binary string                                                                                             |
| <code>_as_hex(str)</code>    | return ref to scalar string containing the value as unsigned hex string, with the '0x' prepended. Leading zeros must be stripped.              |
| <code>_as_bin(str)</code>    | Like <code>_as_hex</code> , only as binary string containing only zeros and ones. Leading zeros must be stripped and a '0b' must be prepended. |
| <code>_rsft(obj,N,B)</code>  | shift object in base B by N 'digits' right<br>For unsupported bases B, return undef to signal failure                                          |
| <code>_lsft(obj,N,B)</code>  | shift object in base B by N 'digits' left<br>For unsupported bases B, return undef to signal failure                                           |
| <code>_xor(obj1,obj2)</code> | XOR (bit-wise) object 1 with object 2<br>Note: XOR, AND and OR pad with zeros if size mismatches                                               |
| <code>_and(obj1,obj2)</code> | AND (bit-wise) object 1 with object 2                                                                                                          |
| <code>_or(obj1,obj2)</code>  | OR (bit-wise) object 1 with object 2                                                                                                           |
| <code>_mod(obj,obj)</code>   | Return remainder of div of the 1st by the 2nd object                                                                                           |
| <code>_sqrt(obj)</code>      | return the square root of object (truncate to int)                                                                                             |
| <code>_fac(obj)</code>       | return factorial of object 1 (1*2*3*4..)                                                                                                       |
| <code>_pow(obj,obj)</code>   | return object 1 to the power of object 2                                                                                                       |
| <code>_gcd(obj,obj)</code>   | return Greatest Common Divisor of two objects                                                                                                  |
| <code>_zeros(obj)</code>     | return number of trailing decimal zeros                                                                                                        |
| <code>_modinv</code>         | return inverse modulus                                                                                                                         |
| <code>_modpow</code>         | return modulus of power (\$x ** \$y) % \$z                                                                                                     |

Input strings come in as unsigned but with prefix (i.e. as '123', '0xabc' or '0b1101').

Testing of input parameter validity is done by the caller, so you need not worry about underflow (f.i. in `_sub()`, `_dec()`) nor about division by zero or similar cases.

The first parameter can be modified, that includes the possibility that you return a reference to a completely different object instead. Although keeping the reference and just changing it's contents is preferred over creating and returning a different reference.

Return values are always references to objects or strings. Exceptions are `_lsft()` and `_rsft()`, which return undef if they can not shift the argument. This is used to delegate shifting of bases different than the one you can support back to `Math::BigInt`, which will use some generic code to calculate the result.

## WRAP YOUR OWN

If you want to port your own favourite c-lib for big numbers to the `Math::BigInt` interface, you can take any of the already existing modules as a rough guideline. You should really wrap up the latest `BigInt` and `BigFloat` testsuites with your module, and replace in them any of the following:

```
use Math::BigInt;
```

by this:

```
use Math::BigInt lib => 'yourlib';
```

This way you ensure that your library really works 100% within `Math::BigInt`.

## LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

**AUTHORS**

Original math code by Mark Biggar, rewritten by Tels <http://bloodgate.com/> in late 2000, 2001. Separated from BigInt and shaped API with the help of John Peacock.

**SEE ALSO**

*[Math::BigInt](#), [Math::BigFloat](#), [Math::BigInt::BitVect](#), [Math::BigInt::GMP](#), [Math::BigInt::Cached](#) and [Math::BigInt::Pari](#).*

**NAME**

Math::BigRat – arbitrarily big rationals

**SYNOPSIS**

```
use Math::BigRat;

$x = Math::BigRat->new('3/7');

print $x->bstr(), "\n";
```

**DESCRIPTION**

This is just a placeholder until the real thing is up and running. Watch this space...

**MATH LIBRARY**

Math with the numbers is done (by default) by a module called Math::BigInt::Calc. This is equivalent to saying:

```
use Math::BigRat lib => 'Calc';
```

You can change this by using:

```
use Math::BigRat lib => 'BitVect';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and when this also fails, revert to Math::BigInt::Calc:

```
use Math::BigRat lib => 'Foo,Math::BigInt::Bar';
```

Calc.pm uses as internal format an array of elements of some decimal base (usually 1e7, but this might be different for some systems) with the least significant digit first, while BitVect.pm uses a bit vector of base 2, most significant bit first. Other modules might use even different means of representing the numbers. See the respective module documentation for further details.

**METHODS**

Any method not listed here is derived from Math::BigFloat (or Math::BigInt), so make sure you check these two modules for further information.

**new()**

```
$x = Math::BigRat->new('1/3');
```

Create a new Math::BigRat object. Input can come in various forms:

```
$x = Math::BigRat->new('1/3'); # simple string
$x = Math::BigRat->new('1 / 3'); # spaced
$x = Math::BigRat->new('1 / 0.1'); # w/ floats
$x = Math::BigRat->new(Math::BigInt->new(3)); # BigInt
$x = Math::BigRat->new(Math::BigFloat->new('3.1')); # BigFloat
$x = Math::BigRat->new(Math::BigInt::Lite->new('2')); # BigLite
```

**numerator()**

```
$n = $x->numerator();
```

Returns a copy of the numerator (the part above the line) as signed BigInt.

**denominator()**

```
$d = $x->denominator();
```

Returns a copy of the denominator (the part under the line) as positive BigInt.

**parts()**

```
($n,$d) = $x->parts();
```

Return a list consisting of (signed) numerator and (unsigned) denominator as BigInts.

**as\_number()**

Returns a copy of the object as BigInt by truncating it to integer.

**bfac()**

```
$x->bfac();
```

Calculates the factorial of \$x. For instance:

```
print Math::BigRat->new('3/1')->bfac(), "\n"; # 1*2*3
print Math::BigRat->new('5/1')->bfac(), "\n"; # 1*2*3*4*5
```

Only works for integers for now.

**blog()**

Is not yet implemented.

**bound()/round()/bround()**

Are not yet implemented.

**BUGS**

Some things are not yet implemented, or only implemented half-way.

**LICENSE**

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

**SEE ALSO**

[Math::BigFloat](#) and [Math::Big](#) as well as [Math::BigInt::BitVect](#), [Math::BigInt::Pari](#) and [Math::BigInt::GMP](#).

The package at <http://search.cpan.org/search?mode=module&query=Math%3A%3ABigRat> may contain more documentation and examples as well as testcases.

**AUTHORS**

(C) by Tels <http://bloodgate.com/> 2001–2002.



**NAME**

Math::Complex – complex numbers and associated mathematical functions

**SYNOPSIS**

```
use Math::Complex;

$z = Math::Complex->make(5, 6);
$t = 4 - 3*i + $z;
$j = cplx(1, 2*pi/3);
```

**DESCRIPTION**

This package lets you create and manipulate complex numbers. By default, *Perl* limits itself to real numbers, but an extra `use` statement brings full complex support, along with a full set of mathematical functions typically associated with and/or extended to complex numbers.

If you wonder what complex numbers are, they were invented to be able to solve the following equation:

$$x^2 = -1$$

and by definition, the solution is noted *i* (engineers use *j* instead since *i* usually denotes an intensity, but the name does not matter). The number *i* is a pure *imaginary* number.

The arithmetics with pure imaginary numbers works just like you would expect it with real numbers... you just have to remember that

$$i*i = -1$$

so you have:

$$\begin{aligned} 5i + 7i &= i * (5 + 7) = 12i \\ 4i - 3i &= i * (4 - 3) = i \\ 4i * 2i &= -8 \\ 6i / 2i &= 3 \\ 1 / i &= -i \end{aligned}$$

Complex numbers are numbers that have both a real part and an imaginary part, and are usually noted:

$$a + bi$$

where *a* is the *real* part and *b* is the *imaginary* part. The arithmetic with complex numbers is straightforward. You have to keep track of the real and the imaginary parts, but otherwise the rules used for real numbers just apply:

$$\begin{aligned} (4 + 3i) + (5 - 2i) &= (4 + 5) + i(3 - 2) = 9 + i \\ (2 + i) * (4 - i) &= 2*4 + 4i - 2i - i*i = 8 + 2i + 1 = 9 + 2i \end{aligned}$$

A graphical representation of complex numbers is possible in a plane (also called the *complex plane*, but it's really a 2D plane). The number

$$z = a + bi$$

is the point whose coordinates are (a, b). Actually, it would be the vector originating from (0, 0) to (a, b). It follows that the addition of two complex numbers is a vectorial addition.

Since there is a bijection between a point in the 2D plane and a complex number (i.e. the mapping is unique and reciprocal), a complex number can also be uniquely identified with polar coordinates:

$$[\text{rho}, \text{theta}]$$

where *rho* is the distance to the origin, and *theta* the angle between the vector and the *x* axis. There is a notation for this using the exponential form, which is:

$$\text{rho} * \exp(i * \text{theta})$$

where  $i$  is the famous imaginary number introduced above. Conversion between this form and the cartesian form  $a + bi$  is immediate:

```
a = rho * cos(theta)
b = rho * sin(theta)
```

which is also expressed by this formula:

```
z = rho * exp(i * theta) = rho * (cos theta + i * sin theta)
```

In other words, it's the projection of the vector onto the  $x$  and  $y$  axes. Mathematicians call *rho* the *norm* or *modulus* and *theta* the *argument* of the complex number. The *norm* of  $z$  will be noted `abs(z)`.

The polar notation (also known as the trigonometric representation) is much more handy for performing multiplications and divisions of complex numbers, whilst the cartesian notation is better suited for additions and subtractions. Real numbers are on the  $x$  axis, and therefore *theta* is zero or  $\pi$ .

All the common operations that can be performed on a real number have been defined to work on complex numbers as well, and are merely *extensions* of the operations defined on real numbers. This means they keep their natural meaning when there is no imaginary part, provided the number is within their definition set.

For instance, the `sqrt` routine which computes the square root of its argument is only defined for non-negative real numbers and yields a non-negative real number (it is an application from  $\mathbf{R}^+$  to  $\mathbf{R}^+$ ). If we allow it to return a complex number, then it can be extended to negative real numbers to become an application from  $\mathbf{R}$  to  $\mathbf{C}$  (the set of complex numbers):

```
sqrt(x) = x >= 0 ? sqrt(x) : sqrt(-x)*i
```

It can also be extended to be an application from  $\mathbf{C}$  to  $\mathbf{C}$ , whilst its restriction to  $\mathbf{R}$  behaves as defined above by using the following definition:

```
sqrt(z = [r,t]) = sqrt(r) * exp(i * t/2)
```

Indeed, a negative real number can be noted  $[x, \pi]$  (the modulus  $x$  is always non-negative, so  $[x, \pi]$  is really  $-x$ , a negative number) and the above definition states that

```
sqrt([x,pi]) = sqrt(x) * exp(i*pi/2) = [sqrt(x),pi/2] = sqrt(x)*i
```

which is exactly what we had defined for negative real numbers above. The `sqrt` returns only one of the solutions: if you want the both, use the `root` function.

All the common mathematical functions defined on real numbers that are extended to complex numbers share that same property of working *as usual* when the imaginary part is zero (otherwise, it would not be called an extension, would it?).

A *new* operation possible on a complex number that is the identity for real numbers is called the *conjugate*, and is noted with a horizontal bar above the number, or  $\sim z$  here.

```
z = a + bi
~z = a - bi
```

Simple... Now look:

```
z * ~z = (a + bi) * (a - bi) = a*a + b*b
```

We saw that the norm of  $z$  was noted `abs(z)` and was defined as the distance to the origin, also known as:

```
rho = abs(z) = sqrt(a*a + b*b)
```

so

```
z * ~z = abs(z) ** 2
```

If  $z$  is a pure real number (i.e.  $b == 0$ ), then the above yields:

```
a * a = abs(a) ** 2
```

which is true (abs has the regular meaning for real number, i.e. stands for the absolute value). This example explains why the norm of  $z$  is noted `abs(z)`: it extends the `abs` function to complex numbers, yet is the regular `abs` we know when the complex number actually has no imaginary part... This justifies *a posteriori* our use of the `abs` notation for the norm.

## OPERATIONS

Given the following notations:

```
z1 = a + bi = r1 * exp(i * t1)
z2 = c + di = r2 * exp(i * t2)
z = <any complex or real number>
```

the following (overloaded) operations are supported on complex numbers:

```
z1 + z2 = (a + c) + i(b + d)
z1 - z2 = (a - c) + i(b - d)
z1 * z2 = (r1 * r2) * exp(i * (t1 + t2))
z1 / z2 = (r1 / r2) * exp(i * (t1 - t2))
z1 ** z2 = exp(z2 * log z1)
~z = a - bi
abs(z) = r1 = sqrt(a*a + b*b)
sqrt(z) = sqrt(r1) * exp(i * t/2)
exp(z) = exp(a) * exp(i * b)
log(z) = log(r1) + i*t
sin(z) = 1/2i (exp(i * z1) - exp(-i * z))
cos(z) = 1/2 (exp(i * z1) + exp(-i * z))
atan2(z1, z2) = atan(z1/z2)
```

The following extra operations are supported on both real and complex numbers:

```
Re(z) = a
Im(z) = b
arg(z) = t
abs(z) = r

cbrt(z) = z ** (1/3)
log10(z) = log(z) / log(10)
logn(z, n) = log(z) / log(n)

tan(z) = sin(z) / cos(z)

csc(z) = 1 / sin(z)
sec(z) = 1 / cos(z)
cot(z) = 1 / tan(z)

asin(z) = -i * log(i*z + sqrt(1-z*z))
acos(z) = -i * log(z + i*sqrt(1-z*z))
atan(z) = i/2 * log((i+z) / (i-z))

acsc(z) = asin(1 / z)
asec(z) = acos(1 / z)
acot(z) = atan(1 / z) = -i/2 * log((i+z) / (z-i))

sinh(z) = 1/2 (exp(z) - exp(-z))
cosh(z) = 1/2 (exp(z) + exp(-z))
tanh(z) = sinh(z) / cosh(z) = (exp(z) - exp(-z)) / (exp(z) + exp(-z))

csch(z) = 1 / sinh(z)
sech(z) = 1 / cosh(z)
coth(z) = 1 / tanh(z)
```

```

asinh(z) = log(z + sqrt(z*z+1))
acosh(z) = log(z + sqrt(z*z-1))
atanh(z) = 1/2 * log((1+z) / (1-z))

acsch(z) = asinh(1 / z)
asech(z) = acosh(1 / z)
acoth(z) = atanh(1 / z) = 1/2 * log((1+z) / (z-1))

```

*arg*, *abs*, *log*, *csc*, *cot*, *acsc*, *acot*, *csch*, *coth*, *acosech*, *acotanh*, have aliases *rho*, *theta*, *ln*, *cosec*, *cotan*, *acosec*, *acotan*, *cosech*, *cotanh*, *acosech*, *acotanh*, respectively. *Re*, *Im*, *arg*, *abs*, *rho*, and *theta* can be used also as mutators. The *cbrt* returns only one of the solutions: if you want all three, use the *root* function.

The *root* function is available to compute all the *n* roots of some complex, where *n* is a strictly positive integer. There are exactly *n* such roots, returned as a list. Getting the number mathematicians call *j* such that:

```
1 + j + j*j = 0;
```

is a simple matter of writing:

```
$j = ((root(1, 3))[1]);
```

The *k*th root for *z* = [*r*,*t*] is given by:

```
(root(z, n))[k] = r**(1/n) * exp(i * (t + 2*k*pi)/n)
```

The *spaceship* comparison operator, *<=>*, is also defined. In order to ensure its restriction to real numbers is conform to what you would expect, the comparison is run on the real part of the complex number first, and imaginary parts are compared only when the real parts match.

## CREATION

To create a complex number, use either:

```
$z = Math::Complex->make(3, 4);
$z = cplx(3, 4);
```

if you know the cartesian form of the number, or

```
$z = 3 + 4*i;
```

if you like. To create a number using the polar form, use either:

```
$z = Math::Complex->emake(5, pi/3);
$x = cplx(5, pi/3);
```

instead. The first argument is the modulus, the second is the angle (in radians, the full circle is *2\*pi*). (Mnemonic: *e* is used as a notation for complex numbers in the polar form).

It is possible to write:

```
$x = cplx(-3, pi/4);
```

but that will be silently converted into [*3*, *-3pi/4*], since the modulus must be non-negative (it represents the distance to the origin in the complex plane).

It is also possible to have a complex number as either argument of the *make*, *emake*, *cplx*, and *cplx**e*: the appropriate component of the argument will be used.

```
$z1 = cplx(-2, 1);
$z2 = cplx($z1, 4);
```

The new, *make*, *emake*, *cplx*, and *cplx**e* will also understand a single (string) argument of the forms

```
2-3i
-3i
```

```
[2, 3]
[2]
```

in which case the appropriate cartesian and exponential components will be parsed from the string and used to create new complex numbers. The imaginary component and the theta, respectively, will default to zero.

## STRINGIFICATION

When printed, a complex number is usually shown under its cartesian style  $a+bi$ , but there are legitimate cases where the polar style  $[r,t]$  is more appropriate.

By calling the class method `Math::Complex::display_format` and supplying either "polar" or "cartesian" as an argument, you override the default display style, which is "cartesian". Not supplying any argument returns the current settings.

This default can be overridden on a per-number basis by calling the `display_format` method instead. As before, not supplying any argument returns the current display style for this number. Otherwise whatever you specify will be the new display style for *this* particular number.

For instance:

```
use Math::Complex;

Math::Complex::display_format('polar');
$j = (root(1, 3))[1];
print "j = $j\n"; # Prints "j = [1,2pi/3]"
$j->display_format('cartesian');
print "j = $j\n"; # Prints "j = -0.5+0.866025403784439i"
```

The polar style attempts to emphasize arguments like  $k\pi/n$  (where  $n$  is a positive integer and  $k$  an integer within  $[-9, +9]$ ), this is called *polar pretty-printing*.

## CHANGED IN PERL 5.6

The `display_format` class method and the corresponding `display_format` object method can now be called using a parameter hash instead of just a one parameter.

The old display format style, which can have values "cartesian" or "polar", can be changed using the "style" parameter.

```
$j->display_format(style => "polar");
```

The one parameter calling convention also still works.

```
$j->display_format("polar");
```

There are two new display parameters.

The first one is "format", which is a `sprintf()`-style format string to be used for both numeric parts of the complex number(s). The is somewhat system-dependent but most often it corresponds to "%.15g". You can revert to the default by setting the format to undef.

```
the $j from the above example

$j->display_format('format' => '%.5f');
print "j = $j\n"; # Prints "j = -0.50000+0.86603i"
$j->display_format('format' => undef);
print "j = $j\n"; # Prints "j = -0.5+0.86603i"
```

Notice that this affects also the return values of the `display_format` methods: in list context the whole parameter hash will be returned, as opposed to only the style parameter value. This is a potential incompatibility with earlier versions if you have been calling the `display_format` method in list context.

The second new display parameter is "polar\_pretty\_print", which can be set to true or false, the default being true. See the previous section for what this means.

**USAGE**

Thanks to overloading, the handling of arithmetics with complex numbers is simple and almost transparent.

Here are some examples:

```
use Math::Complex;

$j = cplx(1, 2*pi/3); # $j ** 3 == 1
print "j = $j, j**3 = ", $j ** 3, "\n";
print "1 + j + j**2 = ", 1 + $j + $j**2, "\n";

$z = -16 + 0*i; # Force it to be a complex
print "sqrt($z) = ", sqrt($z), "\n";

$k = exp(i * 2*pi/3);
print "$j - $k = ", $j - $k, "\n";

$z->Re(3); # Re, Im, arg, abs,
$j->arg(2); # (the last two aka rho, theta)
 # can be used also as mutators.
```

**ERRORS DUE TO DIVISION BY ZERO OR LOGARITHM OF ZERO**

The division (/) and the following functions

|       |       |       |       |
|-------|-------|-------|-------|
| log   | ln    | log10 | logn  |
| tan   | sec   | csc   | cot   |
| atan  | asec  | acsc  | acot  |
| tanh  | sech  | csch  | coth  |
| atanh | asech | acsch | acoth |

cannot be computed for all arguments because that would mean dividing by zero or taking logarithm of zero. These situations cause fatal runtime errors looking like this

```
cot(0): Division by zero.
(Because in the definition of cot(0), the divisor sin(0) is 0)
Died at ...
```

or

```
atanh(-1): Logarithm of zero.
Died at...
```

For the *csc*, *cot*, *asec*, *acsc*, *acot*, *csch*, *coth*, *asech*, *acsch*, the argument cannot be (zero). For the logarithmic functions and the *atanh*, *acoth*, the argument cannot be 1 (one). For the *atanh*, *acoth*, the argument cannot be -1 (minus one). For the *atan*, *acot*, the argument cannot be *i* (the imaginary unit). For the *atan*, *acoth*, the argument cannot be -*i* (the negative imaginary unit). For the *tan*, *sec*, *tanh*, the argument cannot be  $\pi/2 + k * \pi$ , where *k* is any integer.

Note that because we are operating on approximations of real numbers, these errors can happen when merely 'too close' to the singularities listed above.

**ERRORS DUE TO INDIGESTIBLE ARGUMENTS**

The *make* and *emake* accept both real and complex arguments. When they cannot recognize the arguments they will die with error messages like the following

```
Math::Complex::make: Cannot take real part of ...
Math::Complex::make: Cannot take real part of ...
Math::Complex::emake: Cannot take rho of ...
Math::Complex::emake: Cannot take theta of ...
```

## BUGS

Saying `use Math::Complex;` exports many mathematical routines in the caller environment and even overrides some (`sqrt`, `log`). This is construed as a feature by the Authors, actually... :-)

All routines expect to be given real or complex numbers. Don't attempt to use `BigFloat`, since Perl has currently no rule to disambiguate a '+' operation (for instance) between two overloaded entities.

In Cray UNICOS there is some strange numerical instability that results in `root()`, `cos()`, `sin()`, `cosh()`, `sinh()`, losing accuracy fast. Beware. The bug may be in UNICOS math libs, in UNICOS C compiler, in `Math::Complex`. Whatever it is, it does not manifest itself anywhere else where Perl runs.

## AUTHORS

Daniel S. Lewart <*[d-lewart@uiuc.edu](mailto:d-lewart@uiuc.edu)*>

Original authors Raphael Manfredi <*[Raphael\\_Manfredi@pobox.com](mailto:Raphael_Manfredi@pobox.com)*> and Jarkko Hietaniemi <*[jhi@iki.fi](mailto:jhi@iki.fi)*>

**NAME**

Math::Trig – trigonometric functions

**SYNOPSIS**

```
use Math::Trig;

$x = tan(0.9);
$y = acos(3.7);
$z = asin(2.4);

$halfpi = pi/2;

$rad = deg2rad(120);
```

**DESCRIPTION**

Math::Trig defines many trigonometric functions not defined by the core Perl which defines only the `sin()` and `cos()`. The constant **pi** is also defined as are a few convenience functions for angle conversions.

**TRIGONOMETRIC FUNCTIONS**

The tangent

**tan**

The cofunctions of the sine, cosine, and tangent (`cosec/csc` and `cotan/cot` are aliases)

**csc, cosec, sec, sec, cot, cotan**

The arcus (also known as the inverse) functions of the sine, cosine, and tangent

**asin, acos, atan**

The principal value of the arc tangent of y/x

**atan2(y, x)**

The arcus cofunctions of the sine, cosine, and tangent (`acosec/acsc` and `acotan/acot` are aliases)

**acsc, acosec, asec, acot, acotan**

The hyperbolic sine, cosine, and tangent

**sinh, cosh, tanh**

The cofunctions of the hyperbolic sine, cosine, and tangent (`cosech/csch` and `cotanh/coth` are aliases)

**csch, cosech, sech, coth, cotanh**

The arcus (also known as the inverse) functions of the hyperbolic sine, cosine, and tangent

**asinh, acosh, atanh**

The arcus cofunctions of the hyperbolic sine, cosine, and tangent (`acsch/acosech` and `acoth/acotanh` are aliases)

**acsch, acosech, asech, acoth, acotanh**

The trigonometric constant **pi** is also defined.

```
$pi2 = 2 * pi;
```

**ERRORS DUE TO DIVISION BY ZERO**

The following functions

```
acoth
acsc
acsch
```



```

asec
asech
atanh
cot
coth
csc
csch
sec
sech
tan
tanh

```

cannot be computed for all arguments because that would mean dividing by zero or taking logarithm of zero. These situations cause fatal runtime errors looking like this

```

cot(0): Division by zero.
(Because in the definition of cot(0), the divisor sin(0) is 0)
Died at ...

```

or

```

atanh(-1): Logarithm of zero.
Died at...

```

For the `csc`, `cot`, `asec`, `acsc`, `acot`, `csch`, `coth`, `asech`, `acsch`, the argument cannot be (zero). For the `atanh`, `acoth`, the argument cannot be 1 (one). For the `atanh`, `acoth`, the argument cannot be -1 (minus one). For the `tan`, `sec`, `tanh`, `sech`, the argument cannot be  $\pi/2 + k * \pi$ , where  $k$  is any integer.

## SIMPLE (REAL) ARGUMENTS, COMPLEX RESULTS

Please note that some of the trigonometric functions can break out from the **real axis** into the **complex plane**. For example `asin(2)` has no definition for plain real numbers but it has definition for complex numbers.

In Perl terms this means that supplying the usual Perl numbers (also known as scalars, please see [perldata](#)) as input for the trigonometric functions might produce as output results that no more are simple real numbers: instead they are complex numbers.

The `Math::Trig` handles this by using the `Math::Complex` package which knows how to handle complex numbers, please see [Math::Complex](#) for more information. In practice you need not to worry about getting complex numbers as results because the `Math::Complex` takes care of details like for example how to display complex numbers. For example:

```
print asin(2), "\n";
```

should produce something like this (take or leave few last decimals):

```
1.5707963267949-1.31695789692482i
```

That is, a complex number with the real part of approximately 1.571 and the imaginary part of approximately -1.317.

## PLANE ANGLE CONVERSIONS

(Plane, 2-dimensional) angles may be converted with the following functions.

```

$radians = deg2rad($degrees);
$radians = grad2rad($gradians);

$degrees = rad2deg($radians);
$degrees = grad2deg($gradians);

$gradians = deg2grad($degrees);

```

```
$gradians = rad2grad($radians);
```

The full circle is  $2\pi$  radians or 360 degrees or 400 gradians. The result is by default wrapped to be inside the  $[0, [2\pi, 360, 400]]$  circle. If you don't want this, supply a true second argument:

```
$zillions_of_radians = deg2rad($zillions_of_degrees, 1);
$negative_degrees = rad2deg($negative_radians, 1);
```

You can also do the wrapping explicitly by `rad2rad()`, `deg2deg()`, and `grad2grad()`.

## RADIAL COORDINATE CONVERSIONS

**Radial coordinate systems** are the **spherical** and the **cylindrical** systems, explained shortly in more detail.

You can import radial coordinate conversion functions by using the `:radial` tag:

```
use Math::Trig ':radial';

($rho, $theta, $z) = cartesian_to_cylindrical($x, $y, $z);
($rho, $theta, $phi) = cartesian_to_spherical($x, $y, $z);
($x, $y, $z) = cylindrical_to_cartesian($rho, $theta, $z);
($rho_s, $theta, $phi) = cylindrical_to_spherical($rho_c, $theta, $z);
($x, $y, $z) = spherical_to_cartesian($rho, $theta, $phi);
($rho_c, $theta, $z) = spherical_to_cylindrical($rho_s, $theta, $phi);
```

**All angles are in radians.**

## COORDINATE SYSTEMS

**Cartesian** coordinates are the usual rectangular  $(x, y, z)$ -coordinates.

Spherical coordinates,  $(rho, theta, pi)$ , are three-dimensional coordinates which define a point in three-dimensional space. They are based on a sphere surface. The radius of the sphere is **rho**, also known as the *radial* coordinate. The angle in the  $xy$ -plane (around the  $z$ -axis) is **theta**, also known as the *azimuthal* coordinate. The angle from the  $z$ -axis is **phi**, also known as the *polar* coordinate. The 'North Pole' is therefore  $0, 0, rho$ , and the 'Bay of Guinea' (think of the missing big chunk of Africa)  $0, \pi/2, rho$ . In geographical terms  $phi$  is latitude (northward positive, southward negative) and  $theta$  is longitude (eastward positive, westward negative).

**BEWARE:** some texts define *theta* and *phi* the other way round, some texts define the *phi* to start from the horizontal plane, some texts use  $r$  in place of  $rho$ .

Cylindrical coordinates,  $(rho, theta, z)$ , are three-dimensional coordinates which define a point in three-dimensional space. They are based on a cylinder surface. The radius of the cylinder is **rho**, also known as the *radial* coordinate. The angle in the  $xy$ -plane (around the  $z$ -axis) is **theta**, also known as the *azimuthal* coordinate. The third coordinate is the  $z$ , pointing up from the **theta**-plane.

## 3-D ANGLE CONVERSIONS

Conversions to and from spherical and cylindrical coordinates are available. Please notice that the conversions are not necessarily reversible because of the equalities like  $\pi$  angles being equal to  $-\pi$  angles.

**cartesian\_to\_cylindrical**

```
($rho, $theta, $z) = cartesian_to_cylindrical($x, $y, $z);
```

**cartesian\_to\_spherical**

```
($rho, $theta, $phi) = cartesian_to_spherical($x, $y, $z);
```

**cylindrical\_to\_cartesian**

```
($x, $y, $z) = cylindrical_to_cartesian($rho, $theta, $z);
```

**cylindrical\_to\_spherical**

```
($rho_s, $theta, $phi) = cylindrical_to_spherical($rho_c, $theta, $z);
```

Notice that when  $z$  is not 0  $\rho_s$  is not equal to  $\rho_c$ .

spherical\_to\_cartesian

```
($x, $y, $z) = spherical_to_cartesian($rho, $theta, $phi);
```

spherical\_to\_cylindrical

```
($rho_c, $theta, $z) = spherical_to_cylindrical($rho_s, $theta, $phi);
```

Notice that when \$z is not 0 \$rho\_c is not equal to \$rho\_s.

## GREAT CIRCLE DISTANCES AND DIRECTIONS

You can compute spherical distances, called **great circle distances**, by importing the `great_circle_distance()` function:

```
use Math::Trig 'great_circle_distance';

$distance = great_circle_distance($theta0, $phi0, $theta1, $phi1, [, $rho]);
```

The *great circle distance* is the shortest distance between two points on a sphere. The distance is in \$rho units. The \$rho is optional, it defaults to 1 (the unit sphere), therefore the distance defaults to radians.

If you think geographically the *theta* are longitudes: zero at the Greenwich meridian, eastward positive, westward negative—and the *phi* are latitudes: zero at the North Pole, northward positive, southward negative. **NOTE:** this formula thinks in mathematics, not geographically: the *phi* zero is at the North Pole, not at the Equator on the west coast of Africa (Bay of Guinea). You need to subtract your geographical coordinates from  $\pi/2$  (also known as 90 degrees).

```
$distance = great_circle_distance($lon0, pi/2 - $lat0,
 $lon1, pi/2 - $lat1, $rho);
```

The direction you must follow the great circle can be computed by the `great_circle_direction()` function:

```
use Math::Trig 'great_circle_direction';

$direction = great_circle_direction($theta0, $phi0, $theta1, $phi1);
```

The result is in radians, zero indicating straight north,  $\pi$  or  $-\pi$  straight south,  $\pi/2$  straight west, and  $-\pi/2$  straight east.

Notice that the resulting directions might be somewhat surprising if you are looking at a flat worldmap: in such map projections the great circles quite often do not look like the shortest routes—but for example the shortest possible routes from Europe or North America to Asia do often cross the polar regions.

## EXAMPLES

To calculate the distance between London (51.3N 0.5W) and Tokyo (35.7N 139.8E) in kilometers:

```
use Math::Trig qw(great_circle_distance deg2rad);

Notice the 90 - latitude: phi zero is at the North Pole.
@L = (deg2rad(-0.5), deg2rad(90 - 51.3));
@T = (deg2rad(139.8), deg2rad(90 - 35.7));

$km = great_circle_distance(@L, @T, 6378);
```

The direction you would have to go from London to Tokyo

```
use Math::Trig qw(great_circle_direction);

$rad = great_circle_direction(@L, @T);
```

## CAVEAT FOR GREAT CIRCLE FORMULAS

The answers may be off by few percentages because of the irregular (slightly aspherical) form of the Earth. The formula used for great circle distances

```
lat0 = 90 degrees - phi0
```

```
lat1 = 90 degrees - phi1
d = R * arccos(cos(lat0) * cos(lat1) * cos(lon1 - lon01) +
 sin(lat0) * sin(lat1))
```

is also somewhat unreliable for small distances (for locations separated less than about five degrees) because it uses arc cosine which is rather ill-conditioned for values close to zero.

## BUGS

Saying `use Math::Trig;` exports many mathematical routines in the caller environment and even overrides some (`sin`, `cos`). This is construed as a feature by the Authors, actually...;-)

The code is not optimized for speed, especially because we use `Math::Complex` and thus go quite near complex numbers while doing the computations even when the arguments are not. This, however, cannot be completely avoided if we want things like `asin(2)` to give an answer instead of giving a fatal runtime error.

## AUTHORS

Jarkko Hietaniemi <[jhi@iki.fi](mailto:jhi@iki.fi)> and Raphael Manfredi <[Raphael\\_Manfredi@pobox.com](mailto:Raphael_Manfredi@pobox.com)>.

## NAME

Memoize – Make functions faster by trading space for time

## SYNOPSIS

```
This is the documentation for Memoize 1.01
use Memoize;
memoize('slow_function');
slow_function(arguments); # Is faster than it was before
```

This is normally all you need to know. However, many options are available:

```
memoize(function, options...);
```

Options include:

```
NORMALIZER => function
INSTALL => new_name

SCALAR_CACHE => 'MEMORY'
SCALAR_CACHE => ['HASH', \%cache_hash]
SCALAR_CACHE => 'FAULT'
SCALAR_CACHE => 'MERGE'

LIST_CACHE => 'MEMORY'
LIST_CACHE => ['HASH', \%cache_hash]
LIST_CACHE => 'FAULT'
LIST_CACHE => 'MERGE'
```

## DESCRIPTION

'Memoizing' a function makes it faster by trading space for time. It does this by caching the return values of the function in a table. If you call the function again with the same arguments, memoize jumps in and gives you the value out of the table, instead of letting the function compute the value all over again.

Here is an extreme example. Consider the Fibonacci sequence, defined by the following function:

```
Compute Fibonacci numbers
sub fib {
 my $n = shift;
 return $n if $n < 2;
 fib($n-1) + fib($n-2);
}
```

This function is very slow. Why? To compute fib(14), it first wants to compute fib(13) and fib(12), and add the results. But to compute fib(13), it first has to compute fib(12) and fib(11), and then it comes back and computes fib(12) all over again even though the answer is the same. And both of the times that it wants to compute fib(12), it has to compute fib(11) from scratch, and then it has to do it again each time it wants to compute fib(13). This function does so much recomputing of old results that it takes a really long time to run—fib(14) makes 1,200 extra recursive calls to itself, to compute and recompute things that it already computed.

This function is a good candidate for memoization. If you memoize the 'fib' function above, it will compute fib(14) exactly once, the first time it needs to, and then save the result in a table. Then if you ask for fib(14) again, it gives you the result out of the table. While computing fib(14), instead of computing fib(12) twice, it does it once; the second time it needs the value it gets it from the table. It doesn't compute fib(11) four times; it computes it once, getting it from the table the next three times. Instead of making 1,200 recursive calls to 'fib', it makes 15. This makes the function about 150 times faster.

You could do the memoization yourself, by rewriting the function, like this:

```
Compute Fibonacci numbers, memoized version
```

```

{ my @fib;
 sub fib {
 my $n = shift;
 return $fib[$n] if defined $fib[$n];
 return $fib[$n] = $n if $n < 2;
 $fib[$n] = fib($n-1) + fib($n-2);
 }
}

```

Or you could use this module, like this:

```

use Memoize;
memoize('fib');

Rest of the fib function just like the original version.

```

This makes it easy to turn memoizing on and off.

Here's an even simpler example: I wrote a simple ray tracer; the program would look in a certain direction, figure out what it was looking at, and then convert the 'color' value (typically a string like 'red') of that object to a red, green, and blue pixel value, like this:

```

for ($direction = 0; $direction < 300; $direction++) {
 # Figure out which object is in direction $direction
 $color = $object->{color};
 ($r, $g, $b) = @{&ColorToRGB($color)};
 ...
}

```

Since there are relatively few objects in a picture, there are only a few colors, which get looked up over and over again. Memoizing ColorToRGB sped up the program by several percent.

## DETAILS

This module exports exactly one function, `memoize`. The rest of the functions in this package are None of Your Business.

You should say

```
memoize(function)
```

where `function` is the name of the function you want to memoize, or a reference to it. `memoize` returns a reference to the new, memoized version of the function, or `undef` on a non-fatal error. At present, there are no non-fatal errors, but there might be some in the future.

If `function` was the name of a function, then `memoize` hides the old version and installs the new memoized version under the old name, so that `&function(...)` actually invokes the memoized version.

## OPTIONS

There are some optional options you can pass to `memoize` to change the way it behaves a little. To supply options, invoke `memoize` like this:

```

memoize(function, NORMALIZER => function,
 INSTALL => newname,
 SCALAR_CACHE => option,
 LIST_CACHE => option
);

```

Each of these options is optional; you can include some, all, or none of them.

## INSTALL

If you supply a function name with `INSTALL`, `memoize` will install the new, memoized version of the function under the name you give. For example,

```
memoize('fib', INSTALL => 'fastfib')
```

installs the memoized version of `fib` as `fastfib`; without the `INSTALL` option it would have replaced the old `fib` with the memoized version.

To prevent `memoize` from installing the memoized version anywhere, use `INSTALL => undef`.

## NORMALIZER

Suppose your function looks like this:

```
Typical call: f('aha!', A => 11, B => 12);
sub f {
 my $a = shift;
 my %hash = @_;
 $hash{B} ||= 2; # B defaults to 2
 $hash{C} ||= 7; # C defaults to 7

 # Do something with $a, %hash
}
```

Now, the following calls to your function are all completely equivalent:

```
f(OUCH);
f(OUCH, B => 2);
f(OUCH, C => 7);
f(OUCH, B => 2, C => 7);
f(OUCH, C => 7, B => 2);
(etc.)
```

However, unless you tell `Memoize` that these calls are equivalent, it will not know that, and it will compute the values for these invocations of your function separately, and store them separately.

To prevent this, supply a `NORMALIZER` function that turns the program arguments into a string in a way that equivalent arguments turn into the same string. A `NORMALIZER` function for `f` above might look like this:

```
sub normalize_f {
 my $a = shift;
 my %hash = @_;
 $hash{B} ||= 2;
 $hash{C} ||= 7;

 join(',', $a, map ($_ => $hash{$_}) sort keys %hash);
}
```

Each of the argument lists above comes out of the `normalize_f` function looking exactly the same, like this:

```
OUCH,B,2,C,7
```

You would tell `Memoize` to use this normalizer this way:

```
memoize('f', NORMALIZER => 'normalize_f');
```

`memoize` knows that if the normalized version of the arguments is the same for two argument lists, then it can safely look up the value that it computed for one argument list and return it as the result of calling the function with the other argument list, even if the argument lists look different.

The default normalizer just concatenates the arguments with character 28 in between. (In ASCII, this is called FS or control-\.) This always works correctly for functions with only one string argument, and also when the arguments never contain character 28. However, it can confuse certain argument lists:

```
normalizer("a\034", "b")
normalizer("a", "\034b")
```

```
normalizer("a\034\034b")
```

for example.

Since hash keys are strings, the default normalizer will not distinguish between `undef` and the empty string. It also won't work when the function's arguments are references. For example, consider a function `g` which gets two arguments: A number, and a reference to an array of numbers:

```
g(13, [1,2,3,4,5,6,7]);
```

The default normalizer will turn this into something like `"13\034ARRAY(0x436c1f)"`. That would be all right, except that a subsequent array of numbers might be stored at a different location even though it contains the same data. If this happens, Memoize will think that the arguments are different, even though they are equivalent. In this case, a normalizer like this is appropriate:

```
sub normalize { join ' ', $_[0], @{$_[1]} }
```

For the example above, this produces the key `"13 1 2 3 4 5 6 7"`.

Another use for normalizers is when the function depends on data other than those in its arguments. Suppose you have a function which returns a value which depends on the current hour of the day:

```
sub on_duty {
 my ($problem_type) = @_;
 my $hour = (localtime)[2];
 open my $fh, "$DIR/$problem_type" or die...;
 my $line;
 while ($hour-- > 0)
 $line = <$fh>;
 }
 return $line;
}
```

At 10:23, this function generates the 10th line of a data file; at 3:45 PM it generates the 15th line instead. By default, Memoize will only see the `$problem_type` argument. To fix this, include the current hour in the normalizer:

```
sub normalize { join ' ', (localtime)[2], @_ }
```

The calling context of the function (scalar or list context) is propagated to the normalizer. This means that if the memoized function will treat its arguments differently in list context than it would in scalar context, you can have the normalizer function select its behavior based on the results of `wantarray`. Even if called in a list context, a normalizer should still return a single string.

### SCALAR\_CACHE, LIST\_CACHE

Normally, Memoize caches your function's return values into an ordinary Perl hash variable. However, you might like to have the values cached on the disk, so that they persist from one run of your program to the next, or you might like to associate some other interesting semantics with the cached values.

There's a slight complication under the hood of Memoize: There are actually *two* caches, one for scalar values and one for list values. When your function is called in scalar context, its return value is cached in one hash, and when your function is called in list context, its value is cached in the other hash. You can control the caching behavior of both contexts independently with these options.

The argument to `LIST_CACHE` or `SCALAR_CACHE` must either be one of the following four strings:

```
MEMORY
FAULT
MERGE
HASH
```

or else it must be a reference to a list whose first element is one of these four strings, such as `[HASH,`



arguments...].

#### MEMORY

MEMORY means that return values from the function will be cached in an ordinary Perl hash variable. The hash variable will not persist after the program exits. This is the default.

#### HASH

HASH allows you to specify that a particular hash that you supply will be used as the cache. You can tie this hash beforehand to give it any behavior you want.

A tied hash can have any semantics at all. It is typically tied to an on-disk database, so that cached values are stored in the database and retrieved from it again when needed, and the disk file typically persists after your program has exited. See `perltie` for more complete details about `tie`.

A typical example is:

```
use DB_File;
tie my %cache => 'DB_File', $filename, O_RDWR|O_CREAT, 0666;
memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

This has the effect of storing the cache in a `DB_File` database whose name is in `$filename`. The cache will persist after the program has exited. Next time the program runs, it will find the cache already populated from the previous run of the program. Or you can forcibly populate the cache by constructing a batch program that runs in the background and populates the cache file. Then when you come to run your real program the memoized function will be fast because all its results have been precomputed.

**TIE** This option is no longer supported. It is still documented only to aid in the debugging of old programs that use it. Old programs should be converted to use the **HASH** option instead.

```
memoize ... [TIE, PACKAGE, ARGS...]
```

is merely a shortcut for

```
require PACKAGE;
{ my %cache;
 tie %cache, PACKAGE, ARGS...;
}
memoize ... [HASH => \%cache];
```

#### FAULT

**FAULT** means that you never expect to call the function in scalar (or list) context, and that if Memoize detects such a call, it should abort the program. The error message is one of

```
'foo' function called in forbidden list context at line ...
'foo' function called in forbidden scalar context at line ...
```

#### MERGE

**MERGE** normally means the function does not distinguish between list and scalar context, and that return values in both contexts should be stored together. `LIST_CACHE => MERGE` means that list context return values should be stored in the same hash that is used for scalar context returns, and `SCALAR_CACHE => MERGE` means the same, mutatis mutandis. It is an error to specify **MERGE** for both, but it probably does something useful.

Consider this function:

```
sub pi { 3; }
```

Normally, the following code will result in two calls to `pi`:

```
$x = pi();
($y) = pi();
```

```
$z = pi();
```

The first call caches the value 3 in the scalar cache; the second caches the list (3) in the list cache. The third call doesn't call the real `pi` function; it gets the value from the scalar cache.

Obviously, the second call to `pi` is a waste of time, and storing its return value is a waste of space. Specifying `LIST_CACHE => MERGE` will make `memoize` use the same cache for scalar and list context return values, so that the second call uses the scalar cache that was populated by the first call. `pi` ends up being called only once, and both subsequent calls return 3 from the cache, regardless of the calling context.

Another use for `MERGE` is when you want both kinds of return values stored in the same disk file; this saves you from having to deal with two disk files instead of one. You can use a normalizer function to keep the two sets of return values separate. For example:

```
tie my %cache => 'MLDBM', 'DB_File', $filename, ...;

memoize 'myfunc',
 NORMALIZER => 'n',
 SCALAR_CACHE => [HASH => \%cache],
 LIST_CACHE => MERGE,
;

sub n {
 my $context = wantarray() ? 'L' : 'S';
 # ... now compute the hash key from the arguments ...
 $hashkey = "$context:$hashkey";
}
```

This normalizer function will store scalar context return values in the disk file under keys that begin with `S:`, and list context return values under keys that begin with `L:`.

## OTHER FACILITIES

### `unmemoize`

There's an `unmemoize` function that you can import if you want to. Why would you want to? Here's an example: Suppose you have your cache tied to a DBM file, and you want to make sure that the cache is written out to disk if someone interrupts the program. If the program exits normally, this will happen anyway, but if someone types control-C or something then the program will terminate immediately without synchronizing the database. So what you can do instead is

```
$SIG{INT} = sub { unmemoize 'function' };
```

`unmemoize` accepts a reference to, or the name of a previously memoized function, and undoes whatever it did to provide the memoized version in the first place, including making the name refer to the unmemoized version if appropriate. It returns a reference to the unmemoized version of the function.

If you ask it to `unmemoize` a function that was never memoized, it croaks.

### `flush_cache`

`flush_cache(function)` will flush out the caches, discarding *all* the cached data. The argument may be a function name or a reference to a function. For finer control over when data is discarded or expired, see the documentation for `Memoize::Expire`, included in this package.

Note that if the cache is a tied hash, `flush_cache` will attempt to invoke the `CLEAR` method on the hash. If there is no `CLEAR` method, this will cause a run-time error.

An alternative approach to cache flushing is to use the `HASH` option (see above) to request that `Memoize` use a particular hash variable as its cache. Then you can examine or modify the hash at any time in any way you desire. You may flush the cache by using `%hash = ()`.

**CAVEATS**

Memoization is not a cure-all:

- Do not memoize a function whose behavior depends on program state other than its own arguments, such as global variables, the time of day, or file input. These functions will not produce correct results when memoized. For a particularly easy example:

```
sub f {
 time;
}
```

This function takes no arguments, and as far as Memoize is concerned, it always returns the same result. Memoize is wrong, of course, and the memoized version of this function will call `time` once to get the current time, and it will return that same time every time you call it after that.

- Do not memoize a function with side effects.

```
sub f {
 my ($a, $b) = @_;
 my $s = $a + $b;
 print "$a + $b = $s.\n";
}
```

This function accepts two arguments, adds them, and prints their sum. Its return value is the number of characters it printed, but you probably didn't care about that. But Memoize doesn't understand that. If you memoize this function, you will get the result you expect the first time you ask it to print the sum of 2 and 3, but subsequent calls will return 1 (the return value of `print`) without actually printing anything.

- Do not memoize a function that returns a data structure that is modified by its caller.

Consider these functions: `getusers` returns a list of users somehow, and then `main` throws away the first user on the list and prints the rest:

```
sub main {
 my $userlist = getusers();
 shift @$userlist;
 foreach $u (@$userlist) {
 print "User $u\n";
 }
}

sub getusers {
 my @users;
 # Do something to get a list of users;
 \@users; # Return reference to list.
}
```

If you memoize `getusers` here, it will work right exactly once. The reference to the users list will be stored in the memo table. `main` will discard the first element from the referenced list. The next time you invoke `main`, Memoize will not call `getusers`; it will just return the same reference to the same list it got last time. But this time the list has already had its head removed; `main` will erroneously remove another element from it. The list will get shorter and shorter every time you call `main`.

Similarly, this:

```
$u1 = getusers();
$u2 = getusers();
pop @$u1;
```

will modify `$u2` as well as `$u1`, because both variables are references to the same array. Had `getusers` not been memoized, `$u1` and `$u2` would have referred to different arrays.

- Do not memoize a very simple function.

Recently someone mentioned to me that the Memoize module made his program run slower instead of faster. It turned out that he was memoizing the following function:

```
sub square {
 $_[0] * $_[0];
}
```

I pointed out that Memoize uses a hash, and that looking up a number in the hash is necessarily going to take a lot longer than a single multiplication. There really is no way to speed up the square function.

Memoization is not magical.

## PERSISTENT CACHE SUPPORT

You can tie the cache tables to any sort of tied hash that you want to, as long as it supports `TIEHASH`, `FETCH`, `STORE`, and `EXISTS`. For example,

```
tie my %cache => 'GDBM_File', $filename, O_RDWR|O_CREAT, 0666;
memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

works just fine. For some storage methods, you need a little glue.

`SDBM_File` doesn't supply an `EXISTS` method, so included in this package is a glue module called `Memoize::SDBM_File` which does provide one. Use this instead of plain `SDBM_File` to store your cache table on disk in an `SDBM_File` database:

```
tie my %cache => 'Memoize::SDBM_File', $filename, O_RDWR|O_CREAT, 0666;
memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

`NDBM_File` has the same problem and the same solution. (Use `Memoize::NDBM_File` instead of plain `NDBM_File`.)

`Storable` isn't a tied hash class at all. You can use it to store a hash to disk and retrieve it again, but you can't modify the hash while it's on the disk. So if you want to store your cache table in a `Storable` database, use `Memoize::Storable`, which puts a hashlike front-end onto `Storable`. The hash table is actually kept in memory, and is loaded from your `Storable` file at the time you memoize the function, and stored back at the time you unmemoize the function (or when your program exits):

```
tie my %cache => 'Memoize::Storable', $filename;
memoize 'function', SCALAR_CACHE => [HASH => \%cache];

tie my %cache => 'Memoize::Storable', $filename, 'nstore';
memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

Include the `'nstore'` option to have the `Storable` database written in 'network order'. (See [Storable](#) for more details about this.)

The `flush_cache()` function will raise a run-time error unless the tied package provides a `CLEAR` method.

## EXPIRATION SUPPORT

See `Memoize::Expire`, which is a plug-in module that adds expiration functionality to Memoize. If you don't like the kinds of policies that `Memoize::Expire` implements, it is easy to write your own plug-in module to implement whatever policy you desire. Memoize comes with several examples. An expiration manager that implements a LRU policy is available on CPAN as `Memoize::ExpireLRU`.

## BUGS

The test suite is much better, but always needs improvement.

There is some problem with the way `goto &f` works under threaded Perl, perhaps because of the lexical scoping of `@_`. This is a bug in Perl, and until it is resolved, memoized functions will see a slightly different `caller()` and will perform a little more slowly on threaded perls than unthreaded perls.

Some versions of `DB_File` won't let you store data under a key of length 0. That means that if you have a function `f` which you memoized and the cache is in a `DB_File` database, then the value of `f()` (`f` called with no arguments) will not be memoized. If this is a big problem, you can supply a normalizer function that prepends "x" to every key.

## MAILING LIST

To join a very low-traffic mailing list for announcements about Memoize, send an empty note to `mjd-perl-memoize-request@plover.com`.

## AUTHOR

Mark-Jason Dominus (`mjd-perl-memoize+@plover.com`), Plover Systems co.

See the `Memoize.pm` Page at <http://www.plover.com/~mjd/perl/Memoize/> for news and upgrades. Near this page, at <http://www.plover.com/~mjd/perl/MiniMemoize/> there is an article about memoization and about the internals of Memoize that appeared in The Perl Journal, issue #13. (This article is also included in the Memoize distribution as 'article.html'.)

My upcoming book will discuss memoization (and many other fascinating topics) in tremendous detail. It will be published by Morgan Kaufmann in 2002, possibly under the title *Perl Advanced Techniques Handbook*. It will also be available on-line for free. For more information, visit <http://perl.plover.com/book/>.

To join a mailing list for announcements about Memoize, send an empty message to `mjd-perl-memoize-request@plover.com`. This mailing list is for announcements only and has extremely low traffic—about two messages per year.

## COPYRIGHT AND LICENSE

Copyright 1998, 1999, 2000, 2001 by Mark Jason Dominus

This library is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

## THANK YOU

Many thanks to Jonathan Roy for bug reports and suggestions, to Michael Schwern for other bug reports and patches, to Mike Cariaso for helping me to figure out the Right Thing to Do About Expiration, to Joshua Gerth, Joshua Chamas, Jonathan Roy (again), Mark D. Anderson, and Andrew Johnson for more suggestions about expiration, to Brent Powers for the `Memoize::ExpireLRU` module, to Ariel Scolnicov for delightful messages about the Fibonacci function, to Dion Almaer for thought-provoking suggestions about the default normalizer, to Walt Mankowski and Kurt Starsinic for much help investigating problems under threaded Perl, to Alex Dudkevich for reporting the bug in prototyped functions and for checking my patch, to Tony Bass for many helpful suggestions, to Jonathan Roy (again) for finding a use for `unmemoize()`, to Philippe Verdret for enlightening discussion of `Hook::PrePostCall`, to Nat Torkington for advice I ignored, to Chris Nandor for portability advice, to Randal Schwartz for suggesting the 'flush\_cache' function, and to Jenda Krynicky for being a light in the world.

Special thanks to Jarkko Hietaniemi, the 5.8.0 pumpking, for including this module in the core and for his patient and helpful guidance during the integration process.

**NAME**

Memoize::AnyDBM\_File – glue to provide EXISTS for AnyDBM\_File for Storable use

**DESCRIPTION**

See [Memoize](#).

**NAME**

Memoize::Expire – Plug-in module for automatic expiration of memoized values

**SYNOPSIS**

```
use Memoize;
use Memoize::Expire;
tie my %cache => 'Memoize::Expire',
 LIFETIME => $lifetime, # In seconds
 NUM_USES => $n_uses;

memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

**DESCRIPTION**

Memoize::Expire is a plug-in module for Memoize. It allows the cached values for memoized functions to expire automatically. This manual assumes you are already familiar with the Memoize module. If not, you should study that manual carefully first, paying particular attention to the HASH feature.

Memoize::Expire is a layer of software that you can insert in between Memoize itself and whatever underlying package implements the cache. The layer presents a hash variable whose values expire whenever they get too old, have been used too often, or both. You tell Memoize to use this forgetful hash as its cache instead of the default, which is an ordinary hash.

To specify a real-time timeout, supply the LIFETIME option with a numeric value. Cached data will expire after this many seconds, and will be looked up afresh when it expires. When a data item is looked up afresh, its lifetime is reset.

If you specify NUM\_USES with an argument of *n*, then each cached data item will be discarded and looked up afresh after the *n*th time you access it. When a data item is looked up afresh, its number of uses is reset.

If you specify both arguments, data will be discarded from the cache when either expiration condition holds.

Memoize::Expire uses a real hash internally to store the cached data. You can use the HASH option to Memoize::Expire to supply a tied hash in place of the ordinary hash that Memoize::Expire will normally use. You can use this feature to add Memoize::Expire as a layer in between a persistent disk hash and Memoize. If you do this, you get a persistent disk cache whose entries expire automatically. For example:

```
Memoize
|
Memoize::Expire enforces data expiration policy
|
DB_File implements persistence of data in a disk file
|
Disk file

use Memoize;
use Memoize::Expire;
use DB_File;

Set up persistence
tie my %disk_cache => 'DB_File', $filename, O_CREAT|O_RDWR, 0666];

Set up expiration policy, supplying persistent hash as a target
tie my %cache => 'Memoize::Expire',
 LIFETIME => $lifetime, # In seconds
 NUM_USES => $n_uses,
 HASH => \%disk_cache;

Set up memoization, supplying expiring persistent hash for cache
memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

## INTERFACE

There is nothing special about Memoize::Expire. It is just an example. If you don't like the policy that it implements, you are free to write your own expiration policy module that implements whatever policy you desire. Here is how to do that. Let us suppose that your module will be named MyExpirePolicy.

Short summary: You need to create a package that defines four methods:

### TIEHASH

Construct and return cache object.

### EXISTS

Given a function argument, is the corresponding function value in the cache, and if so, is it fresh enough to use?

### FETCH

Given a function argument, look up the corresponding function value in the cache and return it.

### STORE

Given a function argument and the corresponding function value, store them into the cache.

### CLEAR

(Optional.) Flush the cache completely.

The user who wants the memoization cache to be expired according to your policy will say so by writing

```
tie my %cache => 'MyExpirePolicy', args...;
memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

This will invoke `< MyExpirePolicy-TIEHASH(args) . MyExpirePolicy::TIEHASH` should do whatever is appropriate to set up the cache, and it should return the cache object to the caller.

For example, MyExpirePolicy::TIEHASH might create an object that contains a regular Perl hash (which it will to store the cached values) and some extra information about the arguments and how old the data is and things like that. Let us call this object 'C'.

When Memoize needs to check to see if an entry is in the cache already, it will invoke `< C-EXISTS(key) . key` is the normalized function argument. MyExpirePolicy::EXISTS should return 0 if the key is not in the cache, or if it has expired, and 1 if an unexpired value is in the cache. It should *not* return undef, because there is a bug in some versions of Perl that will cause a spurious FETCH if the EXISTS method returns undef.

If your EXISTS function returns true, Memoize will try to fetch the cached value by invoking `< C-FETCH(key) . MyExpirePolicy::FETCH` should return the cached value. Otherwise, Memoize will call the memoized function to compute the appropriate value, and will store it into the cache by calling `< C-STORE(key, value) .`

Here is a very brief example of a policy module that expires each cache item after ten seconds.

```
package Memoize::TenSecondExpire;

sub TIEHASH {
 my ($package, %args) = @_;
 my $cache = $args{HASH} || {};
 bless $cache => $package;
}

sub EXISTS {
 my ($cache, $key) = @_;
 if (exists $cache->{$key} &&
```



```

 $cache->{$key}{EXPIRE_TIME} > time) {
 return 1
 } else {
 return 0; # Do NOT return 'undef' here.
 }
 }

 sub FETCH {
 my ($cache, $key) = @_;
 return $cache->{$key}{VALUE};
 }

 sub STORE {
 my ($cache, $key, $newvalue) = @_;
 $cache->{$key}{VALUE} = $newvalue;
 $cache->{$key}{EXPIRE_TIME} = time + 10;
 }

```

To use this expiration policy, the user would say

```

use Memoize;
tie my %cache10sec => 'Memoize::TenSecondExpire';
memoize 'function', SCALAR_CACHE => [HASH => \%cache10sec];

```

Memoize would then call function whenever a cached value was entirely absent or was older than ten seconds.

You should always support a HASH argument to TIEHASH that ties the underlying cache so that the user can specify that the cache is also persistent or that it has some other interesting semantics. The example above demonstrates how to do this, as does Memoize::Expire.

## ALTERNATIVES

Brent Powers has a Memoize::ExpireLRU module that was designed to work with Memoize and provides expiration of least-recently-used data. The cache is held at a fixed number of entries, and when new data comes in, the least-recently used data is expired. See <http://search.cpan.org/search?mode=module&query=ExpireLRU>.

Joshua Chamas's Tie::Cache module may be useful as an expiration manager. (If you try this, let me know how it works out.)

If you develop any useful expiration managers that you think should be distributed with Memoize, please let me know.

## CAVEATS

This module is experimental, and may contain bugs. Please report bugs to the address below.

Number-of-uses is stored as a 16-bit unsigned integer, so can't exceed 65535.

Because of clock granularity, expiration times may occur up to one second sooner than you expect. For example, suppose you store a value with a lifetime of ten seconds, and you store it at 12:00:00.998 on a certain day. Memoize will look at the clock and see 12:00:00. Then 9.01 seconds later, at 12:00:10.008 you try to read it back. Memoize will look at the clock and see 12:00:10 and conclude that the value has expired. This will probably not occur if you have Time::HiRes installed.

## AUTHOR

Mark-Jason Dominus (mjd-perl-memoize+@plover.com)

Mike Carias provided valuable insight into the best way to solve this problem.

**SEE ALSO**

`perl(1)`

The Memoize man page.

<http://www.plover.com/~mjd/perl/Memoize/> (for news and updates)

I maintain a mailing list on which I occasionally announce new versions of Memoize. The list is for announcements only, not discussion. To join, send an empty message to `mjd-perl-memoize-request@Plover.com`.

**NAME**

Memoize::ExpireFile – test for Memoize expiration semantics

**DESCRIPTION**

See [Memoize::Expire](#).

**NAME**

Memoize::ExpireTest – test for Memoize expiration semantics

**DESCRIPTION**

This module is just for testing expiration semantics. It's not a very good example of how to write an expiration module.

If you are looking for an example, I recommend that you look at the simple example in the Memoize::Expire documentation, or at the code for Memoize::Expire itself.

If you have questions, I will be happy to answer them if you send them to [mjd-perl-memoize+@plover.com](mailto:mjd-perl-memoize+@plover.com).

**NAME**

Memoize::NDBM\_File – glue to provide EXISTS for NDBM\_File for Storable use

**DESCRIPTION**

See [Memoize](#).

**NAME**

Memoize::SDBM\_File – glue to provide EXISTS for SDBM\_File for Storable use

**DESCRIPTION**

See [Memoize](#).

**NAME**

Memoize::Storable – store Memoized data in Storable database

**DESCRIPTION**

See [Memoize](#).

**NAME**

MIME::Base64 – Encoding and decoding of base64 strings

**SYNOPSIS**

```
use MIME::Base64;

$encoded = encode_base64('Aladdin:open sesame');
$decoded = decode_base64($encoded);
```

**DESCRIPTION**

This module provides functions to encode and decode strings into the Base64 encoding specified in RFC 2045 – *MIME (Multipurpose Internet Mail Extensions)*. The Base64 encoding is designed to represent arbitrary sequences of octets in a form that need not be humanly readable. A 65-character subset ([A–Za–z0–9+/=]) of US–ASCII is used, enabling 6 bits to be represented per printable character.

The following functions are provided:

`encode_base64($str, [$eol])`

Encode data by calling the `encode_base64()` function. The first argument is the string to encode. The second argument is the line ending sequence to use (it is optional and defaults to `"\n"`). The returned encoded string is broken into lines of no more than 76 characters each and it will end with `$eol` unless it is empty. Pass an empty string as second argument if you do not want the encoded string broken into lines.

`decode_base64($str)`

Decode a base64 string by calling the `decode_base64()` function. This function takes a single argument which is the string to decode and returns the decoded data.

Any character not part of the 65-character base64 subset set is silently ignored. Characters occurring after a '=' padding character are never decoded.

If the length of the string to decode (after ignoring non-base64 chars) is not a multiple of 4 or padding occurs too early, then a warning is generated if perl is running under `-w`.

If you prefer not to import these routines into your namespace you can call them as:

```
use MIME::Base64 ();
$encoded = MIME::Base64::encode($decoded);
$decoded = MIME::Base64::decode($encoded);
```

**DIAGNOSTICS**

The following warnings might be generated if perl is invoked with the `-w` switch:

**Premature end of base64 data**

The number of characters to decode is not a multiple of 4. Legal base64 data should be padded with one or two "=" characters to make its length a multiple of 4. The decoded result will anyway be as if the padding was there.

**Premature padding of base64 data**

The '=' padding character occurs as the first or second character in a base64 quartet.

**EXAMPLES**

If you want to encode a large file, you should encode it in chunks that are a multiple of 57 bytes. This ensures that the base64 lines line up and that you do not end up with padding in the middle. 57 bytes of data fills one complete base64 line ( $76 == 57 * 4 / 3$ ):

```
use MIME::Base64 qw(encode_base64);

open(FILE, "/var/log/wtmp") or die "$!";
while (read(FILE, $buf, 60*57)) {
```



```
 print encode_base64($buf);
}
```

or if you know you have enough memory

```
use MIME::Base64 qw(encode_base64);
local($/) = undef; # slurp
print encode_base64(<STDIN>);
```

The same approach as a command line:

```
perl -MMIME::Base64 -0777 -ne 'print encode_base64($_)' <file
```

Decoding does not need slurp mode if all the lines contains a multiple of 4 base64 chars:

```
perl -MMIME::Base64 -ne 'print decode_base64($_)' <file
```

## COPYRIGHT

Copyright 1995–1999, 2001 Gisle Aas.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Distantly based on LWP::Base64 written by Martijn Koster <m.koster@nexor.co.uk and Joerg Reichelt <j.reichelt@nexor.co.uk and code posted to comp.lang.perl <3pd2lp\$6gf@wsinti07.win.tue.nl by Hans Mulder <hansm@wsinti07.win.tue.nl

The XS implementation use code from metamail. Copyright 1991 Bell Communications Research, Inc. (Bellcore)

**NAME**

MIME::QuotedPrint – Encoding and decoding of quoted–printable strings

**SYNOPSIS**

```
use MIME::QuotedPrint;

$encoded = encode_qp($decoded);
$decoded = decode_qp($encoded);
```

**DESCRIPTION**

This module provides functions to encode and decode strings into the Quoted–Printable encoding specified in RFC 2045 – *MIME (Multipurpose Internet Mail Extensions)*. The Quoted–Printable encoding is intended to represent data that largely consists of bytes that correspond to printable characters in the ASCII character set. Non–printable characters (as defined by english americans) are represented by a triplet consisting of the character "=" followed by two hexadecimal digits.

The following functions are provided:

`encode_qp($str)`

This function will return an encoded version of the string given as argument.

Note that `encode_qp()` does not change newlines "\n" to the CRLF sequence even though this might be considered the right thing to do (RFC 2045 (Q–P Rule #4)).

`decode_qp($str);`

This function will return the plain text version of the string given as argument.

If you prefer not to import these routines into your namespace you can call them as:

```
use MIME::QuotedPrint ();
$encoded = MIME::QuotedPrint::encode($decoded);
$decoded = MIME::QuotedPrint::decode($encoded);
```

**COPYRIGHT**

Copyright 1995–1997 Gisle Aas.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

**NAME**

Net::Cmd – Network Command class (as used by FTP, SMTP etc)

**SYNOPSIS**

```
use Net::Cmd;

@ISA = qw(Net::Cmd);
```

**DESCRIPTION**

Net::Cmd is a collection of methods that can be inherited by a sub class of IO::Handle. These methods implement the functionality required for a command based protocol, for example FTP and SMTP.

**USER METHODS**

These methods provide a user interface to the Net::Cmd object.

**debug ( VALUE )**

Set the level of debug information for this object. If VALUE is not given then the current state is returned. Otherwise the state is changed to VALUE and the previous state returned.

Different packages may implement different levels of debug but a non-zero value results in copies of all commands and responses also being sent to STDERR.

If VALUE is undef then the debug level will be set to the default debug level for the class.

This method can also be called as a *static* method to set/get the default debug level for a given class.

**message ( )**

Returns the text message returned from the last command

**code ( )**

Returns the 3-digit code from the last command. If a command is pending then the value 0 is returned

**ok ( )**

Returns non-zero if the last code value was greater than zero and less than 400. This holds true for most command servers. Servers where this does not hold may override this method.

**status ( )**

Returns the most significant digit of the current status code. If a command is pending then CMD\_PENDING is returned.

**datasend ( DATA )**

Send data to the remote server, converting LF to CRLF. Any line starting with a '.' will be prefixed with another '.'. DATA may be an array or a reference to an array.

**dataend ( )**

End the sending of data to the remote server. This is done by ensuring that the data already sent ends with CRLF then sending '.CRLF' to end the transmission. Once this data has been sent dataend calls response and returns true if response returns CMD\_OK.

**CLASS METHODS**

These methods are not intended to be called by the user, but used or over-ridden by a sub-class of Net::Cmd

**debug\_print ( DIR, TEXT )**

Print debugging information. DIR denotes the direction *true* being data being sent to the server. Calls debug\_text before printing to STDERR.

**debug\_text ( TEXT )**

This method is called to print debugging information. TEXT is the text being sent. The method should return the text to be printed

This is primarily meant for the use of modules such as FTP where passwords are sent, but we do not want to display them in the debugging information.

**command** ( CMD [, ARGS, ... ] )

Send a command to the command server. All arguments are first joined with a space character and CRLF is appended, this string is then sent to the command server.

Returns undef upon failure

**unsupported** ( )

Sets the status code to 580 and the response text to 'Unsupported command'. Returns zero.

**response** ( )

Obtain a response from the server. Upon success the most significant digit of the status code is returned. Upon failure, timeout etc., *undef* is returned.

**parse\_response** ( TEXT )

This method is called by *response* as a method with one argument. It should return an array of 2 values, the 3-digit status code and a flag which is true when this is part of a multi-line response and this line is not the last.

**getline** ( )

Retrieve one line, delimited by CRLF, from the remote server. Returns *undef* upon failure.

**NOTE:** If you do use this method for any reason, please remember to add some *debug\_print* calls into your method.

**ungetline** ( TEXT )

Unget a line of text from the server.

**read\_until\_dot** ( )

Read data from the remote server until a line consisting of a single '.'. Any lines starting with '.' will have one of the '.'s removed.

Returns a reference to a list containing the lines, or *undef* upon failure.

**tied\_fh** ( )

Returns a filehandle tied to the Net::Cmd object. After issuing a command, you may read from this filehandle using *read()* or *<*. The filehandle will return EOF when the final dot is encountered. Similarly, you may write to the filehandle in order to send data to the server after issuing a command that expects data to be written.

See the Net::POP3 and Net::SMTP modules for examples of this.

## EXPORTS

Net::Cmd exports six subroutines, five of these, CMD\_INFO, CMD\_OK, CMD\_MORE, CMD\_REJECT and CMD\_ERROR, correspond to possible results of response and status. The sixth is CMD\_PENDING.

## AUTHOR

Graham Barr <gbarr@pobox.com>

## COPYRIGHT

Copyright (c) 1995–1997 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=for html <hr

\$Id: //depot/libnet/Net/Cmd.pm#28 \$

**NAME**

Net::Config – Local configuration data for libnet

**SYNOPSIS**

```
use Net::Config qw(%NetConfig);
```

**DESCRIPTION**

Net::Config holds configuration data for the modules in the libnet distribuion. During installation you will be asked for these values.

The configuration data is held globally in a file in the perl installation tree, but a user may override any of these values by providing their own. This can be done by having a `.libnetrc` file in their home directory. This file should return a reference to a HASH containing the keys described below. For example

```
.libnetrc
{
 nntp_hosts => ["my_prefered_host"],
 ph_hosts => ["my_ph_server"],
}
__END__
```

**METHODS**

Net::Config defines the following methods. They are methods as they are invoked as class methods. This is because Net::Config inherits from Net::LocalCfg so you can override these methods if you want.

**requires\_firewall HOST**

Attempts to determine if a given host is outside your firewall. Possible return values are.

```
-1 Cannot lookup hostname
0 Host is inside firewall (or there is no ftp_firewall entry)
1 Host is outside the firewall
```

This is done by using hostname lookup and the `local_netmask` entry in the configuration data.

**NetConfig VALUES**

```
nntp_hosts
snpp_hosts
pop3_hosts
smtp_hosts
ph_hosts
daytime_hosts
time_hosts
```

Each is a reference to an array of hostnames (in order of preference), which should be used for the given protocol

**inet\_domain**

Your internet domain name

**ftp\_firewall**

If you have an FTP proxy firewall (**NOT** an HTTP or SOCKS firewall) then this value should be set to the firewall hostname. If your firewall does not listen to port 21, then this value should be set to "hostname:port" (eg "hostname:99")

**ftp\_firewall\_type**

There are many different ftp firewall products available. But unfortunately there is no standard for how to traverse a firewall. The list below shows the sequence of commands that Net::FTP will use

```

user Username for remote host
pass Password for remote host
fwuser Username for firewall
fwpass Password for firewall
remote.host The hostname of the remote ftp server

```

0    There is no firewall

```

1
 USER user@remote.host
 PASS pass

```

```

2
 USER fwuser
 PASS fwpass
 USER user@remote.host
 PASS pass

```

```

3
 USER fwuser
 PASS fwpass
 SITE remote.site
 USER user
 PASS pass

```

```

4
 USER fwuser
 PASS fwpass
 OPEN remote.site
 USER user
 PASS pass

```

```

5
 USER user@fwuser@remote.site
 PASS pass@fwpass

```

```

6
 USER fwuser@remote.site
 PASS fwpass
 USER user
 PASS pass

```

```

7
 USER user@remote.host
 PASS pass
 AUTH fwuser
 RESP fwpass

```

ftp\_ext\_passive

ftp\_int\_passive

FTP servers normally work on a non-passive mode. That is when you want to transfer data you have to tell the server the address and port to connect to.

With some firewalls this does not work as the server cannot connect to your machine (because you are behind a firewall) and the firewall does not re-write the command. In this case you should set `ftp_ext_passive` to a *true* value.

Some servers are configured to only work in passive mode. If you have one of these you can force

Net::FTP to always transfer in passive mode; when not going via a firewall, by setting `ftp_int_passive` to a *true* value.

#### `local_netmask`

A reference to a list of netmask strings in the form "134.99.4.0/24". These are used by the `requires_firewall` function to determine if a given host is inside or outside your firewall.

The following entries are used during installation & testing on the libnet package

#### `test_hosts`

If true then make `test` may attempt to connect to hosts given in the configuration.

#### `test_exists`

If true then Configure will check each hostname given that it exists

=for html <hr

*\$Id: //depot/libnet/Net/Config.pm#17 \$*

**NAME**

Net::Domain – Attempt to evaluate the current host's internet name and domain

**SYNOPSIS**

```
use Net::Domain qw(hostname hostfqdn hostdomain);
```

**DESCRIPTION**

Using various methods **attempt** to find the Fully Qualified Domain Name (FQDN) of the current host. From this determine the host-name and the host-domain.

Each of the functions will return *undef* if the FQDN cannot be determined.

`hostfqdn ()`

Identify and return the FQDN of the current host.

`hostname ()`

Returns the smallest part of the FQDN which can be used to identify the host.

`hostdomain ()`

Returns the remainder of the FQDN after the *hostname* has been removed.

**AUTHOR**

Graham Barr <gbarr@pobox.com>. Adapted from Sys::Hostname by David Sundstrom  
<sunds@asictest.sc.ti.com>

**COPYRIGHT**

Copyright (c) 1995–1998 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=for html <hr

*\$Id: //depot/libnet/Net/Domain.pm#19 \$*



## NAME

Net::FTP – FTP Client class

## SYNOPSIS

```
use Net::FTP;

$ftp = Net::FTP->new("some.host.name", Debug => 0);
$ftp->login("anonymous", '-anonymous@');
$ftp->cwd("/pub");
$ftp->get("that.file");
$ftp->quit;
```

## DESCRIPTION

Net::FTP is a class implementing a simple FTP client in Perl as described in RFC959. It provides wrappers for a subset of the RFC959 commands.

## OVERVIEW

FTP stands for File Transfer Protocol. It is a way of transferring files between networked machines. The protocol defines a client (whose commands are provided by this module) and a server (not implemented in this module). Communication is always initiated by the client, and the server responds with a message and a status code (and sometimes with data).

The FTP protocol allows files to be sent to or fetched from the server. Each transfer involves a **local file** (on the client) and a **remote file** (on the server). In this module, the same file name will be used for both local and remote if only one is specified. This means that transferring remote file `/path/to/file` will try to put that file in `/path/to/file` locally, unless you specify a local file name.

The protocol also defines several standard **translations** which the file can undergo during transfer. These are ASCII, EBCDIC, binary, and byte. ASCII is the default type, and indicates that the sender of files will translate the ends of lines to a standard representation which the receiver will then translate back into their local representation. EBCDIC indicates the file being transferred is in EBCDIC format. Binary (also known as image) format sends the data as a contiguous bit stream. Byte format transfers the data as bytes, the values of which remain the same regardless of differences in byte size between the two machines (in theory – in practice you should only use this if you really know what you're doing).

## CONSTRUCTOR

`new (HOST [,OPTIONS])`

This is the constructor for a new Net::FTP object. HOST is the name of the remote host to which an FTP connection is required.

OPTIONS are passed in a hash like fashion, using key and value pairs. Possible options are:

**Firewall** – The name of a machine which acts as an FTP firewall. This can be overridden by an environment variable `FTP_FIREWALL`. If specified, and the given host cannot be directly connected to, then the connection is made to the firewall machine and the string `@hostname` is appended to the login identifier. This kind of setup is also referred to as an ftp proxy.

**FirewallType** – The type of firewall running on the machine indicated by **Firewall**. This can be overridden by an environment variable `FTP_FIREWALL_TYPE`. For a list of permissible types, see the description of `ftp_firewall_type` in [Net::Config](#).

**BlockSize** – This is the block size that Net::FTP will use when doing transfers. (defaults to 10240)

**Port** – The port number to connect to on the remote machine for the FTP connection

**Timeout** – Set a timeout value (defaults to 120)

**Debug** – debug level (see the debug method in [Net::Cmd](#))

**Passive** – If set to a non-zero value then all data transfers will be done using passive mode. This is not

usually required except for some *dumb* servers, and some firewall configurations. This can also be set by the environment variable `FTP_PASSIVE`.

**Hash** – If given a reference to a file handle (e.g., `\*STDERR`), print hash marks (#) on that filehandle every 1024 bytes. This simply invokes the `hash()` method for you, so that hash marks are displayed for all transfers. You can, of course, call `hash()` explicitly whenever you'd like.

If the constructor fails `undef` will be returned and an error message will be in `$@`

## METHODS

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

**login** ([LOGIN [,PASSWORD [, ACCOUNT] ] ])

Log into the remote FTP server with the given login information. If no arguments are given then the `Net::FTP` uses the `Net::Netrc` package to lookup the login information for the connected host. If no information is found then a login of *anonymous* is used. If no password is given and the login is *anonymous* then *anonymous@* will be used for password.

If the connection is via a firewall then the `authorize` method will be called with no arguments.

**authorize** ( [AUTH [, RESP]] )

This is a protocol used by some firewall ftp proxies. It is used to authorise the user to send data out. If both arguments are not specified then `authorize` uses `Net::Netrc` to do a lookup.

**site** (ARGS)

Send a SITE command to the remote server and wait for a response.

Returns most significant digit of the response code.

**type** (TYPE [, ARGS])

This method will send the TYPE command to the remote FTP server to change the type of data transfer. The return value is the previous value.

**ascii** ([ARGS]) **binary** ([ARGS]) **ebcdic** ([ARGS]) **byte** ([ARGS])

Synonyms for `type` with the first arguments set correctly

**NOTE** `ebcdic` and `byte` are not fully supported.

**rename** ( OLDNAME, NEWNAME )

Rename a file on the remote FTP server from OLDNAME to NEWNAME. This is done by sending the `RNFR` and `RNTO` commands.

**delete** ( FILENAME )

Send a request to the server to delete FILENAME.

**cwd** ( [ DIR ] )

Attempt to change directory to the directory given in `$dir`. If `$dir` is `".."`, the FTP `CDUP` command is used to attempt to move up one directory. If no directory is given then an attempt is made to change the directory to the root directory.

**cdup** ( )

Change directory to the parent of the current directory.

**pwd** ( )

Returns the full pathname of the current directory.

**restart** ( WHERE )

Set the byte offset at which to begin the next data transfer. `Net::FTP` simply records this value and uses it when during the next data transfer. For this reason this method will not return an error, but setting it may cause a subsequent data transfer to fail.

`rmdir ( DIR )`

Remove the directory with the name `DIR`.

`mkdir ( DIR [, RECURSE ] )`

Create a new directory with the name `DIR`. If `RECURSE` is *true* then `mkdir` will attempt to create all the directories in the given path.

Returns the full pathname to the new directory.

`ls ( [ DIR ] )`

Get a directory listing of `DIR`, or the current directory.

In an array context, returns a list of lines returned from the server. In a scalar context, returns a reference to a list.

`dir ( [ DIR ] )`

Get a directory listing of `DIR`, or the current directory in long format.

In an array context, returns a list of lines returned from the server. In a scalar context, returns a reference to a list.

`get ( REMOTE_FILE [, LOCAL_FILE [, WHERE]] )`

Get `REMOTE_FILE` from the server and store locally. `LOCAL_FILE` may be a filename or a filehandle. If not specified, the file will be stored in the current directory with the same leafname as the remote file.

If `WHERE` is given then the first `WHERE` bytes of the file will not be transferred, and the remaining bytes will be appended to the local file if it already exists.

Returns `LOCAL_FILE`, or the generated local file name if `LOCAL_FILE` is not given. If an error was encountered `undef` is returned.

`put ( LOCAL_FILE [, REMOTE_FILE ] )`

Put a file on the remote server. `LOCAL_FILE` may be a name or a filehandle. If `LOCAL_FILE` is a filehandle then `REMOTE_FILE` must be specified. If `REMOTE_FILE` is not specified then the file will be stored in the current directory with the same leafname as `LOCAL_FILE`.

Returns `REMOTE_FILE`, or the generated remote filename if `REMOTE_FILE` is not given.

**NOTE:** If for some reason the transfer does not complete and an error is returned then the contents that had been transferred will not be removed automatically.

`put_unique ( LOCAL_FILE [, REMOTE_FILE ] )`

Same as `put` but uses the `STOU` command.

Returns the name of the file on the server.

`append ( LOCAL_FILE [, REMOTE_FILE ] )`

Same as `put` but appends to the file on the remote server.

Returns `REMOTE_FILE`, or the generated remote filename if `REMOTE_FILE` is not given.

`unique_name ( )`

Returns the name of the last file stored on the server using the `STOU` command.

`mdtm ( FILE )`

Returns the *modification time* of the given file

`size ( FILE )`

Returns the size in bytes for the given file as stored on the remote server.

**NOTE:** The size reported is the size of the stored file on the remote server. If the file is subsequently transferred from the server in ASCII mode and the remote server and local machine have different ideas about "End Of Line" then the size of file on the local machine after transfer may be different.

`supported ( CMD )`

Returns TRUE if the remote server supports the given command.

`hash ( [FILEHANDLE_GLOB_REF],[BYTES_PER_HASH_MARK] )`

Called without parameters, or with the first argument false, hash marks are suppressed. If the first argument is true but not a reference to a file handle glob, then `\*STDERR` is used. The second argument is the number of bytes per hash mark printed, and defaults to 1024. In all cases the return value is a reference to an array of two: the filehandle glob reference and the bytes per hash mark.

The following methods can return different results depending on how they are called. If the user explicitly calls either of the `pasv` or `port` methods then these methods will return a *true* or *false* value. If the user does not call either of these methods then the result will be a reference to a `Net::FTP::dataconn` based object.

`nlst ( [ DIR ] )`

Send an NLST command to the server, with an optional parameter.

`list ( [ DIR ] )`

Same as `nlst` but using the LIST command

`retr ( FILE )`

Begin the retrieval of a file called FILE from the remote server.

`stor ( FILE )`

Tell the server that you wish to store a file. FILE is the name of the new file that should be created.

`stou ( FILE )`

Same as `stor` but using the STOU command. The name of the unique file which was created on the server will be available via the `unique_name` method after the data connection has been closed.

`appe ( FILE )`

Tell the server that we want to append some data to the end of a file called FILE. If this file does not exist then create it.

If for some reason you want to have complete control over the data connection, this includes generating it and calling the `response` method when required, then the user can use these methods to do so.

However calling these methods only affects the use of the methods above that can return a data connection. They have no effect on methods `get`, `put`, `put_unique` and those that do not require data connections.

`port ( [ PORT ] )`

Send a PORT command to the server. If PORT is specified then it is sent to the server. If not, then a listen socket is created and the correct information sent to the server.

`pasv ( )`

Tell the server to go into passive mode. Returns the text that represents the port on which the server is listening, this text is in a suitable form to sent to another ftp server using the `port` method.

The following methods can be used to transfer files between two remote servers, providing that these two servers can connect directly to each other.

`pasv_xfer ( SRC_FILE, DEST_SERVER [, DEST_FILE ] )`

This method will do a file transfer between two remote ftp servers. If DEST\_FILE is omitted then the leaf name of SRC\_FILE will be used.

`pasv_xfer_unique ( SRC_FILE, DEST_SERVER [, DEST_FILE ] )`

Like `pasv_xfer` but the file is stored on the remote server using the STOU command.

`pasv_wait ( NON_PASV_SERVER )`

This method can be used to wait for a transfer to complete between a passive server and a non-passive server. The method should be called on the passive server with the `Net::FTP` object for the non-passive server passed as an argument.

`abort ( )`

Abort the current data transfer.

`quit ( )`

Send the QUIT command to the remote FTP server and close the socket connection.

### Methods for the adventurous

`Net::FTP` inherits from `Net::Cmd` so methods defined in `Net::Cmd` may be used to send commands to the remote FTP server.

`quot (CMD [,ARGS])`

Send a command, that `Net::FTP` does not directly support, to the remote server and wait for a response.

Returns most significant digit of the response code.

**WARNING** This call should only be used on commands that do not require data connections. Misuse of this method can hang the connection.

### THE dataconn CLASS

Some of the methods defined in `Net::FTP` return an object which will be derived from this class. The `dataconn` class itself is derived from the `IO::Socket::INET` class, so any normal IO operations can be performed. However the following methods are defined in the `dataconn` class and IO should be performed using these.

`read ( BUFFER, SIZE [, TIMEOUT ] )`

Read `SIZE` bytes of data from the server and place it into `BUFFER`, also performing any <CRLF translation necessary. `TIMEOUT` is optional, if not given, the timeout value from the command connection will be used.

Returns the number of bytes read before any <CRLF translation.

`write ( BUFFER, SIZE [, TIMEOUT ] )`

Write `SIZE` bytes of data from `BUFFER` to the server, also performing any <CRLF translation necessary. `TIMEOUT` is optional, if not given, the timeout value from the command connection will be used.

Returns the number of bytes written before any <CRLF translation.

`bytes_read ( )`

Returns the number of bytes read so far.

`abort ( )`

Abort the current data transfer.

`close ( )`

Close the data connection and get a response from the FTP server. Returns *true* if the connection was closed successfully and the first digit of the response from the server was a '2'.

## UNIMPLEMENTED

The following RFC959 commands have not been implemented:

### ALLO

Allocates storage for the file to be transferred.

### SMNT

Mount a different file system structure without changing login or accounting information.

### HELP

Ask the server for "helpful information" (that's what the RFC says) on the commands it accepts.

### MODE

Specifies transfer mode (stream, block or compressed) for file to be transferred.

### SYST

Request remote server system identification.

### STAT

Request remote server status.

### STRU

Specifies file structure for file to be transferred.

### REIN

Reinitialize the connection, flushing all I/O and account information.

## REPORTING BUGS

When reporting bugs/problems please include as much information as possible. It may be difficult for me to reproduce the problem as almost every setup is different.

A small script which yields the problem will probably be of help. It would also be useful if this script was run with the extra options `Debug = 1` passed to the constructor, and the output sent with the bug report. If you cannot include a small script then please include a Debug trace from a run of your program which does yield the problem.

## AUTHOR

Graham Barr <gbarr@pobox.com>

## SEE ALSO

*Net::Netrc* *Net::Cmd*

ftp(1), ftpd(8), RFC 959 <http://www.cis.ohio-state.edu/htbin/rfc/rfc959.html>

## USE EXAMPLES

For an example of the use of `Net::FTP` see

<http://www.csh.rit.edu/~adam/Progs/autoftp-2.0.tar.gz>

`autoftp` is a program that can retrieve, send, or list files via the FTP protocol in a non-interactive manner.

## CREDITS

Henry Gabryjelski <henryg@WPI.EDU> – for the suggestion of creating directories recursively.

Nathan Torkington <gnat@frii.com> – for some input on the documentation.

Roderick Schertler <roderick@gate.net> – for various inputs

**COPYRIGHT**

Copyright (c) 1995–1998 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=for html <hr

*\$Id: //depot/libnet/Net/FTP.pm#68 \$*

**NAME**

Net::hostent – by-name interface to Perl's built-in `gethost*` () functions

**SYNOPSIS**

```
use Net::hostnet;
```

**DESCRIPTION**

This module's default exports override the core `gethostbyname()` and `gethostbyaddr()` functions, replacing them with versions that return "Net::hostent" objects. This object has methods that return the similarly named structure field name from the C's `hostent` structure from *netdb.h*; namely `name`, `aliases`, `addrtype`, `length`, and `addr_list`. The `aliases` and `addr_list` methods return array reference, the rest scalars. The `addr` method is equivalent to the zeroth element in the `addr_list` array reference.

You may also import all the structure fields directly into your namespace as regular variables using the `:FIELDS` import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `h_`. Thus, `$host_obj->name()` corresponds to `$h_name` if you import the fields. Array references are available as regular array variables, so for example `@{$host_obj->aliases() }` would be simply `@h_aliases`.

The `gethost()` function is a simple front-end that forwards a numeric argument to `gethostbyaddr()` by way of `Socket::inet_aton`, and the rest to `gethostbyname()`.

To access this functionality without the core overrides, pass the use an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::pseudo-package`.

**EXAMPLES**

```
use Net::hostent;
use Socket;

@ARGV = ('netscape.com') unless @ARGV;

for $host (@ARGV) {
 unless ($h = gethost($host)) {
 warn "$0: no such host: $host\n";
 next;
 }

 printf "\n%s is %s\n",
 $host,
 lc($h->name) eq lc($host) ? "" : "*really* ",
 $h->name;

 print "\taliases are ", join(", ", @{$h->aliases}), "\n"
 if @{$h->aliases};

 if (@{$h->addr_list} > 1) {
 my $i;
 for $addr (@{$h->addr_list}) {
 printf "\taddr %d is [%s]\n", $i++, inet_ntoa($addr);
 }
 } else {
 printf "\taddress is [%s]\n", inet_ntoa($h->addr);
 }

 if ($h = gethostbyaddr($h->addr)) {
 if (lc($h->name) ne lc($host)) {
 printf "\tThat addr reverses to host %s!\n", $h->name;
 $host = $h->name;
 }
 }
}
```



```
 redo;
 }
}
}
```

**NOTE**

While this class is currently implemented using the `Class::Struct` module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen

**NAME**

libnetFAQ – libnet Frequently Asked Questions

**DESCRIPTION****Where to get this document**

This document is distributed with the libnet distribution, and is also available on the libnet web page at

<http://www.pobox.com/~gbarr/libnet/>

**How to contribute to this document**

You may mail corrections, additions, and suggestions to me [gbarr@pobox.com](mailto:gbarr@pobox.com).

**Author and Copyright Information**

Copyright (c) 1997–1998 Graham Barr. All rights reserved. This document is free; you can redistribute it and/or modify it under the terms of the Artistic License.

**Disclaimer**

This information is offered in good faith and in the hope that it may be of use, but is not guaranteed to be correct, up to date, or suitable for any particular purpose whatsoever. The authors accept no liability in respect of this information or its use.

**Obtaining and installing libnet****What is libnet ?**

libnet is a collection of perl5 modules which all related to network programming. The majority of the modules available provided the client side of popular server–client protocols that are used in the internet community.

**Which version of perl do I need ?**

libnet has been know to work with versions of perl from 5.002 onwards. However if your release of perl is prior to perl5.004 then you will need to obtain and install the IO distribution from CPAN. If you have perl5.004 or later then you will have the IO modules in your installation already, but CPAN may contain updates.

**What other modules do I need ?**

The only modules you will need installed are the modules from the IO distribution. If you have perl5.004 or later you will already have these modules.

**What machines support libnet ?**

libnet itself is an entirely perl–code distribution so it should work on any machine that perl runs on. However IO may not work with some machines and earlier releases of perl. But this should not be the case with perl version 5.004 or later.

**Where can I get the latest libnet release**

The latest libnet release is always on CPAN, you will find it in

<http://www.cpan.org/modules/by-module/Net/>

The latest release and information is also available on the libnet web page at

<http://www.pobox.com/~gbarr/libnet/>

**Using Net::FTP****How do I download files from an FTP server ?**

An example taken from an article posted to comp.lang.perl.misc

```
#!/your/path/to/perl
a module making life easier
```

```

use Net::FTP;

for debugging: $ftp = Net::FTP->new('site','Debug',10);
open a connection and log in!

$ftp = Net::FTP->new('target_site.somewhere.xxx');
$ftp->login('username','password');

set transfer mode to binary

$ftp->binary();

change the directory on the ftp site

$ftp->cwd('/some/path/to/somewhere/');

foreach $name ('file1', 'file2', 'file3') {

get's arguments are in the following order:
ftp server's filename
filename to save the transfer to on the local machine
can be simply used as get($name) if you want the same name

 $ftp->get($name,$name);
}

ftp done!

$ftp->quit;

```

### How do I transfer files in binary mode ?

To transfer files without <LF><CR translation Net::FTP provides the binary method

```
$ftp->binary;
```

### How can I get the size of a file on a remote FTP server ?

### How can I get the modification time of a file on a remote FTP server ?

### How can I change the permissions of a file on a remote server ?

The FTP protocol does not have a command for changing the permissions of a file on the remote server. But some ftp servers may allow a chmod command to be issued via a SITE command, eg

```
$ftp->quot('site chmod 0777',$filename);
```

But this is not guaranteed to work.

### Can I do a reget operation like the ftp command ?

### How do I get a directory listing from an FTP server ?

### Changing directory to "" does not fail ?

Passing an argument of "" to -cwd() has the same affect of calling -cwd() without any arguments. Turn on Debug (*See below*) and you will see what is happening

```

$ftp = Net::FTP->new($host, Debug => 1);
$ftp->login;
$ftp->cwd("");

```

gives

```

Net::FTP=GLOB(0x82196d8)>>> CWD /
Net::FTP=GLOB(0x82196d8)<<< 250 CWD command successful.

```

**I am behind a SOCKS firewall, but the Firewall option does not work ?**

The Firewall option is only for support of one type of firewall. The type supported is an ftp proxy.

To use Net::FTP, or any other module in the libnet distribution, through a SOCKS firewall you must create a socks-ified perl executable by compiling perl with the socks library.

**I am behind an FTP proxy firewall, but cannot access machines outside ?**

Net::FTP implements the most popular ftp proxy firewall approach. The scheme implemented is that where you log in to the firewall with `user@hostname`

I have heard of one other type of firewall which requires a login to the firewall with an account, then a second login with `user@hostname`. You can still use Net::FTP to traverse these firewalls, but a more manual approach must be taken, eg

```
$ftp = Net::FTP->new($firewall) or die $@;
$ftp->login($firewall_user, $firewall_passwd) or die $ftp->message;
$ftp->login($ext_user . '@' . $ext_host, $ext_passwd) or die $ftp->message.
```

**My ftp proxy firewall does not listen on port 21**

FTP servers usually listen on the same port number, port 21, as any other FTP server. But there is no reason why this has to be the case.

If you pass a port number to Net::FTP then it assumes this is the port number of the final destination. By default Net::FTP will always try to connect to the firewall on port 21.

Net::FTP uses IO::Socket to open the connection and IO::Socket allows the port number to be specified as part of the hostname. So this problem can be resolved by either passing a Firewall option like "hostname:1234" or by setting the `ftp_firewall` option in Net::Config to be a string in the same form.

**Is it possible to change the file permissions of a file on an FTP server ?**

The answer to this is "maybe". The FTP protocol does not specify a command to change file permissions on a remote host. However many servers do allow you to run the `chmod` command via the `SITE` command. This can be done with

```
$ftp->site('chmod', '0775', $file);
```

**I have seen scripts call a method message, but cannot find it documented ?**

Net::FTP, like several other packages in libnet, inherits from Net::Cmd, so all the methods described in Net::Cmd are also available on Net::FTP objects.

**Why does Net::FTP not implement mput and mget methods**

The quick answer is because they are easy to implement yourself. The long answer is that to write these in such a way that multiple platforms are supported correctly would just require too much code. Below are some examples how you can implement these yourself.

```
sub mput {
 my($ftp, $pattern) = @_;
 foreach my $file (glob($pattern)) {
 $ftp->put($file) or warn $ftp->message;
 }
}

sub mget {
 my($ftp, $pattern) = @_;
 foreach my $file ($ftp->ls($pattern)) {
 $ftp->get($file) or warn $ftp->message;
 }
}
```

## Using Net::SMTP

### Why can't the part of an Email address after the @ be used as the hostname ?

The part of an Email address which follows the @ is not necessarily a hostname, it is a mail domain. To find the name of a host to connect for a mail domain you need to do a DNS MX lookup

### Why does Net::SMTP not do DNS MX lookups ?

Net::SMTP implements the SMTP protocol. The DNS MX lookup is not part of this protocol.

### The verify method always returns true ?

Well it may seem that way, but it does not. The verify method returns true if the command succeeded. If you pass verify an address which the server would normally have to forward to another machine, the command will succeed with something like

```
252 Couldn't verify <someone@there> but will attempt delivery anyway
```

This command will fail only if you pass it an address in a domain the server directly delivers for, and that address does not exist.

## Debugging scripts

### How can I debug my scripts that use Net::\* modules ?

Most of the libnet client classes allow options to be passed to the constructor, in most cases one option is called Debug. Passing this option with a non-zero value will turn on a protocol trace, which will be sent to STDERR. This trace can be useful to see what commands are being sent to the remote server and what responses are being received back.

```
#!/your/path/to/perl

use Net::FTP;

my $ftp = new Net::FTP($host, Debug => 1);
$ftp->login('gbarr', 'password');
$ftp->quit;
```

this script would output something like

```
Net::FTP: Net::FTP(2.22)
Net::FTP: Exporter
Net::FTP: Net::Cmd(2.0801)
Net::FTP: IO::Socket::INET
Net::FTP: IO::Socket(1.1603)
Net::FTP: IO::Handle(1.1504)

Net::FTP=GLOB(0x8152974)<<< 220 imagine FTP server (Version wu-2.4(5) Tue Jul 29 11:
Net::FTP=GLOB(0x8152974)>>> user gbarr
Net::FTP=GLOB(0x8152974)<<< 331 Password required for gbarr.
Net::FTP=GLOB(0x8152974)>>> PASS
Net::FTP=GLOB(0x8152974)<<< 230 User gbarr logged in. Access restrictions apply.
Net::FTP=GLOB(0x8152974)>>> QUIT
Net::FTP=GLOB(0x8152974)<<< 221 Goodbye.
```

The first few lines tell you the modules that Net::FTP uses and their versions, this is useful data to me when a user reports a bug. The last seven lines show the communication with the server. Each line has three parts. The first part is the object itself, this is useful for separating the output if you are using multiple objects. The second part is either <<<< to show data coming from the server or >>>> to show data going to the server. The remainder of the line is the command being sent or response being received.

**AUTHOR AND COPYRIGHT**

Copyright (c) 1997 Graham Barr. All rights reserved.

=for html <hr

*\$Id: //depot/libnet/Net/libnetFAQ.pod#5 \$*

**NAME**

Net::netent – by-name interface to Perl's built-in `getnet*` () functions

**SYNOPSIS**

```
use Net::netent qw(:FIELDS);
getnetbyname("loopback") or die "bad net";
printf "%s is %08X\n", $n_name, $n_net;

use Net::netent;

$n = getnetbyname("loopback") or die "bad net";
{ # there's gotta be a better way, eh?
 @bytes = unpack("C4", pack("N", $n->net));
 shift @bytes while @bytes && $bytes[0] == 0;
}
printf "%s is %08X [%d.%d.%d.%d]\n", $n->name, $n->net, @bytes;
```

**DESCRIPTION**

This module's default exports override the core `getnetbyname()` and `getnetbyaddr()` functions, replacing them with versions that return "Net::netent" objects. This object has methods that return the similarly named structure field name from the C's netent structure from *netdb.h*; namely `name`, `aliases`, `addrtype`, and `net`. The `aliases` method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the `:FIELDS` import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `n_`. Thus, `$net_obj->name()` corresponds to `$n_name` if you import the fields. Array references are available as regular array variables, so for example `@{ $net_obj->aliases() }` would be simply `@n_aliases`.

The `getnet()` function is a simple front-end that forwards a numeric argument to `getnetbyaddr()`, and the rest to `getnetbyname()`.

To access this functionality without the core overrides, pass the use an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::pseudo-package`.

**EXAMPLES**

The `getnet()` functions do this in the Perl core:

```
sv_setiv(sv, (I32)nent->n_net);
```

The `gethost()` functions do this in the Perl core:

```
sv_setpv(sv, hent->h_addr, len);
```

That means that the address comes back in binary for the host functions, and as a regular perl integer for the net ones. This seems a bug, but here's how to deal with it:

```
use strict;
use Socket;
use Net::netent;

@ARGV = ('loopback') unless @ARGV;
my($n, $net);
for $net (@ARGV) {
 unless ($n = getnetbyname($net)) {
 warn "$0: no such net: $net\n";
 next;
 }
}
```

```

 printf "\n%s is %s%s\n",
 $net,
 lc($n->name) eq lc($net) ? "" : "*really* ",
 $n->name;

 print "\taliases are ", join(", ", @{$n->aliases}), "\n"
 if @{$n->aliases};

 # this is stupid; first, why is this not in binary?
 # second, why am i going through these convolutions
 # to make it looks right
 {
 my @a = unpack("C4", pack("N", $n->net));
 shift @a while @a && $a[0] == 0;
 printf "\taddr is %s [%d.%d.%d.%d]\n", $n->net, @a;
 }

 if ($n = getnetbyaddr($n->net)) {
 if (lc($n->name) ne lc($net)) {
 printf "\tThat addr reverses to net %s!\n", $n->name;
 $net = $n->name;
 redo;
 }
 }
}

```

**NOTE**

While this class is currently implemented using the `Class::Struct` module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen



**NAME**

Net::Netrc – OO interface to users netrc file

**SYNOPSIS**

```
use Net::Netrc;

$mach = Net::Netrc->lookup('some.machine');
$login = $mach->login;
($login, $password, $account) = $mach->lpa;
```

**DESCRIPTION**

Net::Netrc is a class implementing a simple interface to the .netrc file used as by the ftp program.

Net::Netrc also implements security checks just like the ftp program, these checks are, first that the .netrc file must be owned by the user and second the ownership permissions should be such that only the owner has read and write access. If these conditions are not met then a warning is output and the .netrc file is not read.

**THE .netrc FILE**

The .netrc file contains login and initialization information used by the auto-login process. It resides in the user's home directory. The following tokens are recognized; they may be separated by spaces, tabs, or new-lines:

**machine name**

Identify a remote machine name. The auto-login process searches the .netrc file for a machine token that matches the remote machine specified. Once a match is made, the subsequent .netrc tokens are processed, stopping when the end of file is reached or another machine or a default token is encountered.

**default**

This is the same as machine name except that default matches any name. There can be only one default token, and it must be after all machine tokens. This is normally used as:

```
default login anonymous password user@site
```

thereby giving the user automatic anonymous login to machines not specified in .netrc.

**login name**

Identify a user on the remote machine. If this token is present, the auto-login process will initiate a login using the specified name.

**password string**

Supply a password. If this token is present, the auto-login process will supply the specified string if the remote server requires a password as part of the login process.

**account string**

Supply an additional account password. If this token is present, the auto-login process will supply the specified string if the remote server requires an additional account password.

**macdef name**

Define a macro. Net::Netrc only parses this field to be compatible with *ftp*.

**CONSTRUCTOR**

The constructor for a Net::Netrc object is not called new as it does not really create a new object. But instead is called lookup as this is essentially what it does.

**lookup ( MACHINE [, LOGIN ])**

Lookup and return a reference to the entry for MACHINE. If LOGIN is given then the entry returned will have the given login. If LOGIN is not given then the first entry in the .netrc file for MACHINE will

be returned.

If a matching entry cannot be found, and a default entry exists, then a reference to the default entry is returned.

If there is no matching entry found and there is no default defined, or no .netrc file is found, then undef is returned.

## METHODS

login ( )

Return the login id for the netrc entry

password ( )

Return the password for the netrc entry

account ( )

Return the account information for the netrc entry

lpa ( )

Return a list of login, password and account information for the netrc entry

## AUTHOR

Graham Barr <gbarr@pobox.com>

## SEE ALSO

*[Net::Netrc](#) [Net::Cmd](#)*

## COPYRIGHT

Copyright (c) 1995–1998 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=for html <hr

\$Id: //depot/libnet/Net/Netrc.pm#13 \$

**NAME**

Net::NNTP – NNTP Client class

**SYNOPSIS**

```
use Net::NNTP;

$nnntp = Net::NNTP->new("some.host.name");
$nnntp->quit;
```

**DESCRIPTION**

Net::NNTP is a class implementing a simple NNTP client in Perl as described in RFC977. Net::NNTP inherits its communication methods from Net::Cmd

**CONSTRUCTOR**

`new ( [ HOST ] [, OPTIONS ] )`

This is the constructor for a new Net::NNTP object. HOST is the name of the remote host to which a NNTP connection is required. If not given two environment variables are checked, first NNTPSERVER then NEWSHOST, then Net::Config is checked, and if a host is not found then news is used.

OPTIONS are passed in a hash like fashion, using key and value pairs. Possible options are:

**Timeout** – Maximum time, in seconds, to wait for a response from the NNTP server, a value of zero will cause all IO operations to block. (default: 120)

**Debug** – Enable the printing of debugging information to STDERR

**Reader** – If the remote server is INN then initially the connection will be to nnrpd, by default Net::NNTP will issue a MODE READER command so that the remote server becomes innd. If the Reader option is given with a value of zero, then this command will not be sent and the connection will be left talking to nnrpd.

**METHODS**

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

`article ( [ MSGID|MSGNUM ], [FH] )`

Retrieve the header, a blank line, then the body (text) of the specified article.

If FH is specified then it is expected to be a valid filehandle and the result will be printed to it, on success a true value will be returned. If FH is not specified then the return value, on success, will be a reference to an array containing the article requested, each entry in the array will contain one line of the article.

If no arguments are passed then the current article in the currently selected newsgroup is fetched.

MSGNUM is a numeric id of an article in the current newsgroup, and will change the current article pointer. MSGID is the message id of an article as shown in that article's header. It is anticipated that the client will obtain the MSGID from a list provided by the newnews command, from references contained within another article, or from the message-id provided in the response to some other commands.

If there is an error then *undef* will be returned.

`body ( [ MSGID|MSGNUM ], [FH] )`

Like `article` but only fetches the body of the article.

`head ( [ MSGID|MSGNUM ], [FH] )`

Like `article` but only fetches the headers for the article.

`articlefh ( [ MSGID|MSGNUM ] )`  
`bodyfh ( [ MSGID|MSGNUM ] )`  
`headfh ( [ MSGID|MSGNUM ] )`

These are similar to `article()`, `body()` and `head()`, but rather than returning the requested data directly, they return a tied filehandle from which to read the article.

`nntpstat ( [ MSGID|MSGNUM ] )`

The `nntpstat` command is similar to the `article` command except that no text is returned. When selecting by message number within a group, the `nntpstat` command serves to set the "current article pointer" without sending text.

Using the `nntpstat` command to select by message-id is valid but of questionable value, since a selection by message-id does **not** alter the "current article pointer".

Returns the message-id of the "current article".

`group ( [ GROUP ] )`

Set and/or get the current group. If `GROUP` is not given then information is returned on the current group.

In a scalar context it returns the group name.

In an array context the return value is a list containing, the number of articles in the group, the number of the first article, the number of the last article and the group name.

`ihave ( MSGID [, MESSAGE ] )`

The `ihave` command informs the server that the client has an article whose id is `MSGID`. If the server desires a copy of that article, and `MESSAGE` has been given the it will be sent.

Returns *true* if the server desires the article and `MESSAGE` was successfully sent, if specified.

If `MESSAGE` is not specified then the message must be sent using the `datasend` and `dataend` methods from *Net::Cmd*

`MESSAGE` can be either an array of lines or a reference to an array.

`last ( )`

Set the "current article pointer" to the previous article in the current newsgroup.

Returns the message-id of the article.

`date ( )`

Returns the date on the remote server. This date will be in a UNIX time format (seconds since 1970)

`postok ( )`

`postok` will return *true* if the servers initial response indicated that it will allow posting.

`authinfo ( USER, PASS )`

`list ( )`

Obtain information about all the active newsgroups. The results is a reference to a hash where the key is a group name and each value is a reference to an array. The elements in this array are:– the last article number in the group, the first article number in the group and any information flags about the group.

`newgroups ( SINCE [, DISTRIBUTIONS ] )`

`SINCE` is a time value and `DISTRIBUTIONS` is either a distribution pattern or a reference to a list of distribution patterns. The result is the same as `list`, but the groups return will be limited to those created after `SINCE` and, if specified, in one of the distribution areas in `DISTRIBUTIONS`.

`newnews ( SINCE [, GROUPS [, DISTRIBUTIONS ]])`

`SINCE` is a time value. `GROUPS` is either a group pattern or a reference to a list of group patterns. `DISTRIBUTIONS` is either a distribution pattern or a reference to a list of distribution patterns.

Returns a reference to a list which contains the message-ids of all news posted after `SINCE`, that are in a groups which matched `GROUPS` and a distribution which matches `DISTRIBUTIONS`.

`next ( )`

Set the "current article pointer" to the next article in the current newsgroup.

Returns the message-id of the article.

`post ( [ MESSAGE ] )`

Post a new article to the news server. If `MESSAGE` is specified and posting is allowed then the message will be sent.

If `MESSAGE` is not specified then the message must be sent using the `datasend` and `dataend` methods from [Net::Cmd](#)

`MESSAGE` can be either an array of lines or a reference to an array.

`postfh ( )`

Post a new article to the news server using a tied filehandle. If posting is allowed, this method will return a tied filehandle that you can `print ( )` the contents of the article to be posted. You must explicitly `close ( )` the filehandle when you are finished posting the article, and the return value from the `close ( )` call will indicate whether the message was successfully posted.

`slave ( )`

Tell the remote server that I am not a user client, but probably another news server.

`quit ( )`

Quit the remote server and close the socket connection.

### Extension methods

These methods use commands that are not part of the RFC977 documentation. Some servers may not support all of them.

`newsgroups ( [ PATTERN ] )`

Returns a reference to a hash where the keys are all the group names which match `PATTERN`, or all of the groups if no pattern is specified, and each value contains the description text for the group.

`distributions ( )`

Returns a reference to a hash where the keys are all the possible distribution names and the values are the distribution descriptions.

`subscriptions ( )`

Returns a reference to a list which contains a list of groups which are recommended for a new user to subscribe to.

`overview_fmt ( )`

Returns a reference to an array which contain the names of the fields returned by `xover`.

`active_times ( )`

Returns a reference to a hash where the keys are the group names and each value is a reference to an array containing the time the groups was created and an identifier, possibly an Email address, of the creator.

**active ( [ PATTERN ] )**

Similar to `list` but only active groups that match the pattern are returned. `PATTERN` can be a group pattern.

**xgtitle ( PATTERN )**

Returns a reference to a hash where the keys are all the group names which match `PATTERN` and each value is the description text for the group.

**xhdr ( HEADER, MESSAGE-SPEC )**

Obtain the header field `HEADER` for all the messages specified.

The return value will be a reference to a hash where the keys are the message numbers and each value contains the text of the requested header for that message.

**xover ( MESSAGE-SPEC )**

The return value will be a reference to a hash where the keys are the message numbers and each value contains a reference to an array which contains the overview fields for that message.

The names of the fields can be obtained by calling `overview_fmt`.

**xpath ( MESSAGE-ID )**

Returns the path name to the file on the server which contains the specified message.

**xpat ( HEADER, PATTERN, MESSAGE-SPEC )**

The result is the same as `xhdr` except the is will be restricted to headers where the text of the header matches `PATTERN`

**xrover**

The `XROVER` command returns reference information for the article(s) specified.

Returns a reference to a `HASH` where the keys are the message numbers and the values are the References: lines from the articles

**listgroup ( [ GROUP ] )**

Returns a reference to a list of all the active messages in `GROUP`, or the current group if `GROUP` is not specified.

**reader**

Tell the server that you are a reader and not another server.

This is required by some servers. For example if you are connecting to an INN server and you have transfer permission your connection will be connected to the transfer daemon, not the NNTP daemon. Issuing this command will cause the transfer daemon to hand over control to the NNTP daemon.

Some servers do not understand this command, but issuing it and ignoring the response is harmless.

**UNSUPPORTED**

The following NNTP command are unsupported by the package, and there are no plans to do so.

AUTHINFO GENERIC  
XTHREAD  
XSEARCH  
XINDEX

**DEFINITIONS****MESSAGE-SPEC**

`MESSAGE-SPEC` is either a single message-id, a single message number, or a reference to a list of two message numbers.

If MESSAGE-SPEC is a reference to a list of two message numbers and the second number in a range is less than or equal to the first then the range represents all messages in the group after the first message number.

**NOTE** For compatibility reasons only with earlier versions of Net::NNTP a message spec can be passed as a list of two numbers, this is deprecated and a reference to the list should now be passed

## PATTERN

The NNTP protocol uses the WILDMAT format for patterns. The WILDMAT format was first developed by Rich Salz based on the format used in the UNIX "find" command to articulate file names. It was developed to provide a uniform mechanism for matching patterns in the same manner that the UNIX shell matches filenames.

Patterns are implicitly anchored at the beginning and end of each string when testing for a match.

There are five pattern matching operations other than a strict one-to-one match between the pattern and the source to be checked for a match.

The first is an asterisk \* to match any sequence of zero or more characters.

The second is a question mark ? to match any single character. The third specifies a specific set of characters.

The set is specified as a list of characters, or as a range of characters where the beginning and end of the range are separated by a minus (or dash) character, or as any combination of lists and ranges. The dash can also be included in the set as a character if it is the beginning or end of the set. This set is enclosed in square brackets. The close square bracket ] may be used in a set if it is the first character in the set.

The fourth operation is the same as the logical not of the third operation and is specified the same way as the third with the addition of a caret character ^ at the beginning of the test string just inside the open square bracket.

The final operation uses the backslash character to invalidate the special meaning of an open square bracket [, the asterisk, backslash or the question mark. Two backslashes in sequence will result in the evaluation of the backslash as a character with no special meaning.

### Examples

[^] -]

matches any single character other than a close square bracket or a minus sign/dash.

\*bdc

matches any string that ends with the string "bdc" including the string "bdc" (without quotes).

[0-9a-zA-Z]

matches any single printable alphanumeric ASCII character.

a???d

matches any four character string which begins with a and ends with d.

## SEE ALSO

[Net::Cmd](#)

## AUTHOR

Graham Barr <gbarr@pobox.com>

## COPYRIGHT

Copyright (c) 1995-1997 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=for html <hr

*\$Id: //depot/libnet/Net/NNTP.pm#15 \$*



**NAME**

Net::Ping – check a remote host for reachability

\$Id: Ping.pm,v 1.6 2002/06/19 15:23:48 rob Exp \$

**SYNOPSIS**

```
use Net::Ping;

$p = Net::Ping->new();
print "$host is alive.\n" if $p->ping($host);
$p->close();

$p = Net::Ping->new("icmp");
$p->bind($my_addr); # Specify source interface of pings
foreach $host (@host_array)
{
 print "$host is ";
 print "NOT " unless $p->ping($host, 2);
 print "reachable.\n";
 sleep(1);
}
$p->close();

$p = Net::Ping->new("tcp", 2);
Try connecting to the www port instead of the echo port
$p->{port_num} = getservbyname("http", "tcp");
while ($stop_time > time())
{
 print "$host not reachable ", scalar(localtime()), "\n"
 unless $p->ping($host);
 sleep(300);
}
undef($p);

High precision syntax (requires Time::HiRes)
$p = Net::Ping->new();
$p->hires();
($ret, $duration, $ip) = $p->ping($host, 5.5);
printf("$host [ip: $ip] is alive (packet return time: %.2f ms)\n", 1000 * $duration)
 if $ret;
$p->close();

For backward compatibility
print "$host is alive.\n" if pingecho($host);
```

**DESCRIPTION**

This module contains methods to test the reachability of remote hosts on a network. A ping object is first created with optional parameters, a variable number of hosts may be pinged multiple times and then the connection is closed.

You may choose one of four different protocols to use for the ping. The "udp" protocol is the default. Note that a live remote host may still fail to be pingable by one or more of these protocols. For example, [www.microsoft.com](http://www.microsoft.com) is generally alive but not pingable.

With the "tcp" protocol the `ping()` method attempts to establish a connection to the remote host's echo port. If the connection is successfully established, the remote host is considered reachable. No data is actually echoed. This protocol does not require any special privileges but has higher overhead than the other two protocols.

Specifying the "udp" protocol causes the `ping()` method to send a udp packet to the remote host's echo port. If the echoed packet is received from the remote host and the received packet contains the same data as the packet that was sent, the remote host is considered reachable. This protocol does not require any special privileges. It should be borne in mind that, for a udp ping, a host will be reported as unreachable if it is not running the appropriate echo service. For Unix-like systems see [inetd\(8\)](#) for more information.

If the "icmp" protocol is specified, the `ping()` method sends an icmp echo message to the remote host, which is what the UNIX ping program does. If the echoed message is received from the remote host and the echoed information is correct, the remote host is considered reachable. Specifying the "icmp" protocol requires that the program be run as root or that the program be setuid to root.

If the "external" protocol is specified, the `ping()` method attempts to use the `Net::Ping::External` module to ping the remote host. `Net::Ping::External` interfaces with your system's default ping utility to perform the ping, and generally produces relatively accurate results. If `Net::Ping::External` is not installed on your system, specifying the "external" protocol will result in an error.

## Functions

`Net::Ping->new([ $proto [, $def_timeout [, $bytes]] ] );`

Create a new ping object. All of the parameters are optional. `$proto` specifies the protocol to use when doing a ping. The current choices are "tcp", "udp" or "icmp". The default is "udp".

If a default timeout (`$def_timeout`) in seconds is provided, it is used when a timeout is not given to the `ping()` method (below). The timeout must be greater than 0 and the default, if not specified, is 5 seconds.

If the number of data bytes (`$bytes`) is given, that many data bytes are included in the ping packet sent to the remote host. The number of data bytes is ignored if the protocol is "tcp". The minimum (and default) number of data bytes is 1 if the protocol is "udp" and 0 otherwise. The maximum number of data bytes that can be specified is 1024.

`$p->ping($host [, $timeout] );`

Ping the remote host and wait for a response. `$host` can be either the hostname or the IP number of the remote host. The optional timeout must be greater than 0 seconds and defaults to whatever was specified when the ping object was created. Returns a success flag. If the hostname cannot be found or there is a problem with the IP number, the success flag returned will be undef. Otherwise, the success flag will be 1 if the host is reachable and 0 if it is not. For most practical purposes, undef and 0 and can be treated as the same case. In array context, the elapsed time is also returned. The elapsed time value will be a float, as returned by the `Time::HiRes::time()` function, if `hires()` has been previously called, otherwise it is returned as an integer.

`$p->source_verify( { 0 | 1 } );`

Allows source endpoint verification to be enabled or disabled. This is useful for those remote destinations with multiples interfaces where the response may not originate from the same endpoint that the original destination endpoint was sent to. This only affects udp and icmp protocol pings.

This is enabled by default.

`$p->hires( { 0 | 1 } );`

Causes this module to use `Time::HiRes` module, allowing milliseconds to be returned by subsequent calls to `ping()`.

This is disabled by default.

`$p->bind($local_addr);`

Sets the source address from which pings will be sent. This must be the address of one of the interfaces on the local host. `$local_addr` may be specified as a hostname or as a text IP address such as "192.168.1.1".

If the protocol is set to "tcp", this method may be called any number of times, and each call to the

`ping()` method (below) will use the most recent `$local_addr`. If the protocol is "icmp" or "udp", then `bind()` must be called at most once per object, and (if it is called at all) must be called before the first call to `ping()` for that object.

`$p-open($host);`

When you are using the stream protocol, this call pre-opens the tcp socket. It's only necessary to do this if you want to provide a different timeout when creating the connection, or remove the overhead of establishing the connection from the first ping. If you don't call `open()`, the connection is automatically opened the first time `ping()` is called. This call simply does nothing if you are using any protocol other than stream.

`$p-close();`

Close the network connection for this ping object. The network connection is also closed by "undef `$p`". The network connection is automatically closed if the ping object goes out of scope (e.g. `$p` is local to a subroutine and you leave the subroutine).

`pingecho($host [, $timeout]);`

To provide backward compatibility with the previous version of Net::Ping, a `pingecho()` subroutine is available with the same functionality as before. `pingecho()` uses the tcp protocol. The return values and parameters are the same as described for the `ping()` method. This subroutine is obsolete and may be removed in a future version of Net::Ping.

## WARNING

`pingecho()` or a ping object with the tcp protocol use `alarm()` to implement the timeout. So, don't use `alarm()` in your program while you are using `pingecho()` or a ping object with the tcp protocol. The udp and icmp protocols do not use `alarm()` to implement the timeout.

## NOTES

There will be less network overhead (and some efficiency in your program) if you specify either the udp or the icmp protocol. The tcp protocol will generate 2.5 times or more traffic for each ping than either udp or icmp. If many hosts are pinged frequently, you may wish to implement a small wait (e.g. 25ms or more) between each ping to avoid flooding your network with packets.

The icmp protocol requires that the program be run as root or that it be setuid to root. The other protocols do not require special privileges, but not all network devices implement tcp or udp echo.

Local hosts should normally respond to pings within milliseconds. However, on a very congested network it may take up to 3 seconds or longer to receive an echo packet from the remote host. If the timeout is set too low under these conditions, it will appear that the remote host is not reachable (which is almost the truth).

Reachability doesn't necessarily mean that the remote host is actually functioning beyond its ability to echo packets. tcp is slightly better at indicating the health of a system than icmp because it uses more of the networking stack to respond.

Because of a lack of anything better, this module uses its own routines to pack and unpack ICMP packets. It would be better for a separate module to be written which understands all of the different kinds of ICMP packets.

## INSTALL

The latest source tree is available via cvs:

```
cvs -z3 -q -d :pserver:anonymous@cvs.roobik.com.: /usr/local/cvsroot/freeware co Net
cd Net-Ping
```

The tarball can be created as follows:

```
perl Makefile.PL ; make ; make dist
```

The latest Net::Ping release can be found at CPAN:

```
$CPAN/modules/by-module/Net/
```

1) Extract the tarball

```
gtar -zxvf Net-Ping-xxxx.tar.gz
cd Net-Ping-xxxx
```

2) Build:

```
make realclean
perl Makefile.PL
make
make test
```

3) Install

```
make install
```

Or install it RPM Style:

```
rpm -ta SOURCES/Net-Ping-xxxx.tar.gz
rpm -ih RPMS/noarch/perl-Net-Ping-xxxx.rpm
```

## AUTHORS

Current maintainer:

bbb@cpan.org (Rob Brown)

External protocol:

colinm@cpan.org (Colin McMillen)

Stream protocol:

bronson@trestle.com (Scott Bronson)

Original pingecho():

karrer@bernina.ethz.ch (Andreas Karrer)

pmarquess@bfsec.bt.co.uk (Paul Marquess)

Original Net::Ping author:

mose@ns.ccsn.edu (Russell Mosemann)

## COPYRIGHT

Copyright (c) 2002, Rob Brown. All rights reserved.

Copyright (c) 2001, Colin McMillen. All rights reserved.

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

## NAME

Net::POP3 – Post Office Protocol 3 Client class (RFC1939)

## SYNOPSIS

```
use Net::POP3;

Constructors
$pop = Net::POP3->new('pop3host');
$pop = Net::POP3->new('pop3host', Timeout => 60);
```

## DESCRIPTION

This module implements a client interface to the POP3 protocol, enabling a perl5 application to talk to POP3 servers. This documentation assumes that you are familiar with the POP3 protocol described in RFC1939.

A new Net::POP3 object must be created with the *new* method. Once this has been done, all POP3 commands are accessed via method calls on the object.

## EXAMPLES

Need some small examples in here :-)

## CONSTRUCTOR

`new ( [ HOST, ] [ OPTIONS ] )`

This is the constructor for a new Net::POP3 object. HOST is the name of the remote host to which a POP3 connection is required.

If HOST is not given, then the POP3\_Host specified in Net::Config will be used.

OPTIONS are passed in a hash like fashion, using key and value pairs. Possible options are:

**ResvPort** – If given then the socket for the Net::POP3 object will be bound to the local port given using *bind* when the socket is created.

**Timeout** – Maximum time, in seconds, to wait for a response from the POP3 server (default: 120)

**Debug** – Enable debugging information

## METHODS

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

`user ( USER )`

Send the USER command.

`pass ( PASS )`

Send the PASS command. Returns the number of messages in the mailbox.

`login ( [ USER [, PASS ] ] )`

Send both the USER and PASS commands. If PASS is not given the Net::POP3 uses Net::Netrc to lookup the password using the host and username. If the username is not specified then the current user name will be used.

Returns the number of messages in the mailbox. However if there are no messages on the server the string "0E0" will be returned. This will give a true value in a boolean context, but zero in a numeric context.

If there was an error authenticating the user then *undef* will be returned.

`apop ( [ USER [, PASS ] ] )`

Authenticate with the server identifying as USER with password PASS. Similar to *login*, but the password is not sent in clear text.

To use this method you must have the `Digest::MD5` or the `MD5` module installed, otherwise this method will return *undef*.

`top ( MSGNUM [, NUMLINES ] )`

Get the header and the first `NUMLINES` of the body for the message `MSGNUM`. Returns a reference to an array which contains the lines of text read from the server.

`list ( [ MSGNUM ] )`

If called with an argument the `list` returns the size of the message in octets.

If called without arguments a reference to a hash is returned. The keys will be the `MSGNUM`'s of all undeleted messages and the values will be their size in octets.

`get ( MSGNUM [, FH ] )`

Get the message `MSGNUM` from the remote mailbox. If `FH` is not given then `get` returns a reference to an array which contains the lines of text read from the server. If `FH` is given then the lines returned from the server are printed to the filehandle `FH`.

`getfh ( MSGNUM )`

As per `get ( )`, but returns a tied filehandle. Reading from this filehandle returns the requested message. The filehandle will return EOF at the end of the message and should not be reused.

`last ( )`

Returns the highest `MSGNUM` of all the messages accessed.

`popstat ( )`

Returns a list of two elements. These are the number of undeleted elements and the size of the mbox in octets.

`ping ( USER )`

Returns a list of two elements. These are the number of new messages and the total number of messages for `USER`.

`uidl ( [ MSGNUM ] )`

Returns a unique identifier for `MSGNUM` if given. If `MSGNUM` is not given `uidl` returns a reference to a hash where the keys are the message numbers and the values are the unique identifiers.

`delete ( MSGNUM )`

Mark message `MSGNUM` to be deleted from the remote mailbox. All messages that are marked to be deleted will be removed from the remote mailbox when the server connection closed.

`reset ( )`

Reset the status of the remote POP3 server. This includes resetting the status of all messages to not be deleted.

`quit ( )`

Quit and close the connection to the remote POP3 server. Any messages marked as deleted will be deleted from the remote mailbox.

## NOTES

If a `Net::POP3` object goes out of scope before `quit` method is called then the `reset` method will be called before the connection is closed. This means that any messages marked to be deleted will not be.

## SEE ALSO

*[Net::Netrc](#), [Net::Cmd](#)*

**AUTHOR**

Graham Barr <gbarr@pobox.com>

**COPYRIGHT**

Copyright (c) 1995–1997 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=for html <hr

*\$Id: //depot/libnet/Net/POP3.pm#22 \$*

## NAME

Net::protoent – by-name interface to Perl's built-in `getproto*` () functions

## SYNOPSIS

```
use Net::protoent;
$p = getprotobyname(shift || 'tcp') || die "no proto";
printf "proto for %s is %d, aliases are %s\n",
 $p->name, $p->proto, "@{$p->aliases}";

use Net::protoent qw(:FIELDS);
getprotobyname(shift || 'tcp') || die "no proto";
print "proto for $p_name is $p_proto, aliases are @p_aliases\n";
```

## DESCRIPTION

This module's default exports override the core `getprotoent()`, `getprotobyname()`, and `getnetbyport()` functions, replacing them with versions that return "Net::protoent" objects. They take default second arguments of "tcp". This object has methods that return the similarly named structure field name from the C's protoent structure from *netdb.h*; namely `name`, `aliases`, and `proto`. The `aliases` method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the `:FIELDS` import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `p_`. Thus, `$proto_obj->name()` corresponds to `$p_name` if you import the fields. Array references are available as regular array variables, so for example `@{$proto_obj->aliases() }` would be simply `@p_aliases`.

The `getproto()` function is a simple front-end that forwards a numeric argument to `getprotobyport()`, and the rest to `getprotobyname()`.

To access this functionality without the core overrides, pass the use an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

## NOTE

While this class is currently implemented using the `Class::Struct` module to build a struct-like class, you shouldn't rely upon this.

## AUTHOR

Tom Christiansen



**NAME**

Net::servent – by-name interface to Perl's built-in `getserv*()` functions

**SYNOPSIS**

```
use Net::servent;
$s = getservbyname(shift || 'ftp') || die "no service";
printf "port for %s is %s, aliases are %s\n",
 $s->name, $s->port, "@{$s->aliases}";

use Net::servent qw(:FIELDS);
getservbyname(shift || 'ftp') || die "no service";
print "port for $s_name is $s_port, aliases are @s_aliases\n";
```

**DESCRIPTION**

This module's default exports override the core `getservent()`, `getservbyname()`, and `getnetbyport()` functions, replacing them with versions that return "Net::servent" objects. They take default second arguments of "tcp". This object has methods that return the similarly named structure field name from the C's servent structure from *netdb.h*; namely name, aliases, port, and proto. The aliases method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the `:FIELDS` import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `n_`. Thus, `$serv_obj->name()` corresponds to `$s_name` if you import the fields. Array references are available as regular array variables, so for example `@{$serv_obj->aliases() }` would be simply `@s_aliases`.

The `getserv()` function is a simple front-end that forwards a numeric argument to `getnetbyport()`, and the rest to `getservbyname()`.

To access this functionality without the core overrides, pass the use an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

**EXAMPLES**

```
use Net::servent qw(:FIELDS);

while (@ARGV) {
 my ($service, $proto) = ((split m!/!, shift), 'tcp');
 my $valet = getserv($service, $proto);
 unless ($valet) {
 warn "$0: No service: $service/$proto\n";
 next;
 }
 printf "service $service/$proto is port %d\n", $valet->port;
 print "alias are @s_aliases\n" if @s_aliases;
}
```

**NOTE**

While this class is currently implemented using the `Class::Struct` module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen

## NAME

Net::SMTP – Simple Mail Transfer Protocol Client

## SYNOPSIS

```
use Net::SMTP;

Constructors
$smtp = Net::SMTP->new('mailhost');
$smtp = Net::SMTP->new('mailhost', Timeout => 60);
```

## DESCRIPTION

This module implements a client interface to the SMTP and ESMTP protocol, enabling a perl5 application to talk to SMTP servers. This documentation assumes that you are familiar with the concepts of the SMTP protocol described in RFC821.

A new Net::SMTP object must be created with the *new* method. Once this has been done, all SMTP commands are accessed through this object.

The Net::SMTP class is a subclass of Net::Cmd and IO::Socket::INET.

## EXAMPLES

This example prints the mail domain name of the SMTP server known as mailhost:

```
#!/usr/local/bin/perl -w

use Net::SMTP;

$smtp = Net::SMTP->new('mailhost');
print $smtp->domain, "\n";
$smtp->quit;
```

This example sends a small message to the postmaster at the SMTP server known as mailhost:

```
#!/usr/local/bin/perl -w

use Net::SMTP;

$smtp = Net::SMTP->new('mailhost');

$smtp->mail($ENV{USER});
$smtp->to('postmaster');

$smtp->data();
$smtp->datasend("To: postmaster\n");
$smtp->datasend("\n");
$smtp->datasend("A simple test message\n");
$smtp->dataend();

$smtp->quit;
```

## CONSTRUCTOR

**new** Net::SMTP [ HOST, ] [ OPTIONS ]

This is the constructor for a new Net::SMTP object. HOST is the name of the remote host to which an SMTP connection is required.

If HOST is not given, then the SMTP\_Host specified in Net::Config will be used.

OPTIONS are passed in a hash like fashion, using key and value pairs. Possible options are:

**Hello** – SMTP requires that you identify yourself. This option specifies a string to pass as your mail domain. If not given a guess will be taken.

**LocalAddr** and **LocalPort** – These parameters are passed directly to IO::Socket to allow binding the

socket to a local port.

**Timeout** – Maximum time, in seconds, to wait for a response from the SMTP server (default: 120)

**Debug** – Enable debugging information

Example:

```
$smtp = Net::SMTP->new('mailhost',
 Hello => 'my.mail.domain',
 Timeout => 30,
 Debug => 1,
);
```

## METHODS

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

**banner ( )**

Returns the banner message which the server replied with when the initial connection was made.

**domain ( )**

Returns the domain that the remote SMTP server identified itself as during connection.

**hello ( DOMAIN )**

Tell the remote server the mail domain which you are in using the EHLO command (or HELO if EHLO fails). Since this method is invoked automatically when the Net::SMTP object is constructed the user should normally not have to call it manually.

**etrn ( DOMAIN )**

Request a queue run for the DOMAIN given.

**auth ( USERNAME, PASSWORD )**

Attempt SASL authentication.

**mail ( ADDRESS [, OPTIONS] )**

**send ( ADDRESS )**

**send\_or\_mail ( ADDRESS )**

**send\_and\_mail ( ADDRESS )**

Send the appropriate command to the server MAIL, SEND, SOML or SAML. ADDRESS is the address of the sender. This initiates the sending of a message. The method *recipient* should be called for each address that the message is to be sent to.

The mail method can some additional ESMTP OPTIONS which is passed in hash like fashion, using key and value pairs. Possible options are:

```
Size => <bytes>
Return => <??>
Bits => "7" | "8"
Transaction => <ADDRESS>
Envelope => <ENVID>
```

**reset ( )**

Reset the status of the server. This may be called after a message has been initiated, but before any data has been sent, to cancel the sending of the message.

**recipient ( ADDRESS [, ADDRESS [ ...]] [, OPTIONS] )**

Notify the server that the current message should be sent to all of the addresses given. Each address is sent as a separate command to the server. Should the sending of any address result in a failure then the process is aborted and a *false* value is returned. It is up to the user to call *reset* if they so desire.

The `recipient` method can some additional `OPTIONS` which is passed in hash like fashion, using key and value pairs. Possible options are:

```
Notify =>
SkipBad => ignore bad addresses
```

If `SkipBad` is true the `recipient` will not return an error when a bad address is encountered and it will return an array of addresses that did succeed.

```
$smtp->recipient($recipient1,$recipient2); # Good
$smtp->recipient($recipient1,$recipient2, { SkipBad => 1 }); # Good
$smtp->recipient("$recipient,$recipient2"); # BAD
```

```
to (ADDRESS [, ADDRESS [...]])
cc (ADDRESS [, ADDRESS [...]])
bcc (ADDRESS [, ADDRESS [...]])
```

Synonyms for `recipient`.

```
data ([DATA])
```

Initiate the sending of the data from the current message.

`DATA` may be a reference to a list or a list. If specified the contents of `DATA` and a termination string `".\r\n"` is sent to the server. And the result will be true if the data was accepted.

If `DATA` is not specified then the result will indicate that the server wishes the data to be sent. The data must then be sent using the `datasend` and `dataend` methods described in [Net::Cmd](#).

```
expand (ADDRESS)
```

Request the server to expand the given address Returns an array which contains the text read from the server.

```
verify (ADDRESS)
```

Verify that `ADDRESS` is a legitimate mailing address.

```
help ([$subject])
```

Request help text from the server. Returns the text or undef upon failure

```
quit ()
```

Send the `QUIT` command to the remote SMTP server and close the socket connection.

## ADDRESSES

All methods that accept addresses expect the address to be a valid rfc2821-quoted address, although `Net::SMTP` will accept accept the address surrounded by angle brackets.

```
funny user@domain WRONG
"funny user"@domain RIGHT, recommended
<"funny user"@domain> OK
```

## SEE ALSO

[Net::Cmd](#)

## AUTHOR

Graham Barr <gbarr@pobox.com>

## COPYRIGHT

Copyright (c) 1995–1997 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=for html <hr

`$Id: //depot/libnet/Net/SMTP.pm#25 $`

**NAME**

Net::Time – time and daytime network client interface

**SYNOPSIS**

```
use Net::Time qw(inet_time inet_daytime);

print inet_time(); # use default host from Net::Config
print inet_time('localhost');
print inet_time('localhost', 'tcp');

print inet_daytime(); # use default host from Net::Config
print inet_daytime('localhost');
print inet_daytime('localhost', 'tcp');
```

**DESCRIPTION**

Net::Time provides subroutines that obtain the time on a remote machine.

`inet_time ( [HOST [, PROTOCOL [, TIMEOUT]]])`

Obtain the time on `HOST`, or some default host if `HOST` is not given or not defined, using the protocol as defined in RFC868. The optional argument `PROTOCOL` should define the protocol to use, either `tcp` or `udp`. The result will be a time value in the same units as returned by `time()` or *undef* upon failure.

`inet_daytime ( [HOST [, PROTOCOL [, TIMEOUT]]])`

Obtain the time on `HOST`, or some default host if `HOST` is not given or not defined, using the protocol as defined in RFC867. The optional argument `PROTOCOL` should define the protocol to use, either `tcp` or `udp`. The result will be an ASCII string or *undef* upon failure.

**AUTHOR**

Graham Barr <gbarr@pobox.com>

**COPYRIGHT**

Copyright (c) 1995–1998 Graham Barr. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=for html <hr

*\$Id: //depot/libnet/Net/Time.pm#9 \$*

**NAME**

NEXT.pm – Provide a pseudo-class NEXT that allows method redispatch

**SYNOPSIS**

```
use NEXT;

package A;
sub A::method { print "$_[0]: A method\n"; $_[0]->NEXT::method() }
sub A::DESTROY { print "$_[0]: A dtor\n"; $_[0]->NEXT::DESTROY() }

package B;
use base qw(A);
sub B::AUTOLOAD { print "$_[0]: B AUTOLOAD\n"; $_[0]->NEXT::AUTOLOAD() }
sub B::DESTROY { print "$_[0]: B dtor\n"; $_[0]->NEXT::DESTROY() }

package C;
sub C::method { print "$_[0]: C method\n"; $_[0]->NEXT::method() }
sub C::AUTOLOAD { print "$_[0]: C AUTOLOAD\n"; $_[0]->NEXT::AUTOLOAD() }
sub C::DESTROY { print "$_[0]: C dtor\n"; $_[0]->NEXT::DESTROY() }

package D;
use base qw(B C);
sub D::method { print "$_[0]: D method\n"; $_[0]->NEXT::method() }
sub D::AUTOLOAD { print "$_[0]: D AUTOLOAD\n"; $_[0]->NEXT::AUTOLOAD() }
sub D::DESTROY { print "$_[0]: D dtor\n"; $_[0]->NEXT::DESTROY() }

package main;

my $obj = bless {}, "D";

$obj->method(); # Calls D::method, A::method, C::method
$obj->missing_method(); # Calls D::AUTOLOAD, B::AUTOLOAD, C::AUTOLOAD

Clean-up calls D::DESTROY, B::DESTROY, A::DESTROY, C::DESTROY
```

**DESCRIPTION**

NEXT.pm adds a pseudoclass named NEXT to any program that uses it. If a method `m` calls `$self-NEXT::m()`, the call to `m` is redispatched as if the calling method had not originally been found.

In other words, a call to `$self-NEXT::m()` resumes the depth-first, left-to-right search of `$self`'s class hierarchy that resulted in the original call to `m`.

Note that this is not the same thing as `$self-SUPER::m()`, which begins a new dispatch that is restricted to searching the ancestors of the current class. `$self-NEXT::m()` can backtrack past the current class — to look for a suitable method in other ancestors of `$self` — whereas `$self-SUPER::m()` cannot.

A typical use would be in the destructors of a class hierarchy, as illustrated in the synopsis above. Each class in the hierarchy has a DESTROY method that performs some class-specific action and then redispatches the call up the hierarchy. As a result, when an object of class D is destroyed, the destructors of *all* its parent classes are called (in depth-first, left-to-right order).

Another typical use of redispatch would be in AUTOLOAD'ed methods. If such a method determined that it was not able to handle a particular call, it might choose to redispatch that call, in the hope that some other AUTOLOAD (above it, or to its left) might do better.

By default, if a redispatch attempt fails to find another method elsewhere in the objects class hierarchy, it quietly gives up and does nothing (but see *"Enforcing redispatch"*). This gracious acquiescence is also unlike the (generally annoying) behaviour of SUPER, which throws an exception if it cannot redispatch.

Note that it is a fatal error for any method (including AUTOLOAD) to attempt to redispatch any method that does not have the same name. For example:

```
sub D::oops { print "oops!\n"; $_[0]->NEXT::other_method() }
```

### Enforcing redispach

It is possible to make NEXT redispach more demandingly (i.e. like SUPER does), so that the redispach throws an exception if it cannot find a "next" method to call.

To do this, simply invoke the redispach as:

```
$self->NEXT::ACTUAL::method();
```

rather than:

```
$self->NEXT::method();
```

The ACTUAL tells NEXT that there must actually be a next method to call, or it should throw an exception.

NEXT::ACTUAL is most commonly used in AUTOLOAD methods, as a means to decline an AUTOLOAD request, but preserve the normal exception-on-failure semantics:

```
sub AUTOLOAD {
 if ($AUTOLOAD =~ /foo|bar/) {
 # handle here
 }
 else { # try elsewhere
 shift->NEXT::ACTUAL::AUTOLOAD(@_);
 }
}
```

By using NEXT::ACTUAL, if there is no other AUTOLOAD to handle the method call, an exception will be thrown (as usually happens in the absence of a suitable AUTOLOAD).

### Avoiding repetitions

If NEXT redispaching is used in the methods of a "diamond" class hierarchy:

```
A B
/ \ /
C D
\ /
E

use NEXT;

package A;
sub foo { print "called A::foo\n"; shift->NEXT::foo() }

package B;
sub foo { print "called B::foo\n"; shift->NEXT::foo() }

package C; @ISA = qw(A);
sub foo { print "called C::foo\n"; shift->NEXT::foo() }

package D; @ISA = qw(A B);
sub foo { print "called D::foo\n"; shift->NEXT::foo() }

package E; @ISA = qw(C D);
sub foo { print "called E::foo\n"; shift->NEXT::foo() }

E->foo();
```

then derived classes may (re-)inherit base-class methods through two or more distinct paths (e.g. in the way E inherits A::foo twice — through C and D). In such cases, a sequence of NEXT redispaches will invoke the multiply inherited method as many times as it is inherited. For example, the above code prints:

```
called E::foo
```

```
called C::foo
called A::foo
called D::foo
called A::foo
called B::foo
```

(i.e. `A::foo` is called twice).

In some cases this *may* be the desired effect within a diamond hierarchy, but in others (e.g. for destructors) it may be more appropriate to call each method only once during a sequence of redispaches.

To cover such cases, you can redispach methods via:

```
$self->NEXT::UNSEEN::method();
```

rather than:

```
$self->NEXT::method();
```

This causes the dispatcher to skip any classes in the hierarchy that it has already visited in an earlier redispach. So, for example, if the previous example were rewritten:

```
package A;
sub foo { print "called A::foo\n"; shift->NEXT::UNSEEN::foo() }

package B;
sub foo { print "called B::foo\n"; shift->NEXT::UNSEEN::foo() }

package C; @ISA = qw(A);
sub foo { print "called C::foo\n"; shift->NEXT::UNSEEN::foo() }

package D; @ISA = qw(A B);
sub foo { print "called D::foo\n"; shift->NEXT::UNSEEN::foo() }

package E; @ISA = qw(C D);
sub foo { print "called E::foo\n"; shift->NEXT::UNSEEN::foo() }

E->foo();
```

then it would print:

```
called E::foo
called C::foo
called A::foo
called D::foo
called B::foo
```

and omit the second call to `A::foo`.

Note that you can also use:

```
$self->NEXT::UNSEEN::ACTUAL::method();
```

or:

```
$self->NEXT::ACTUAL::UNSEEN::method();
```

to get both unique invocation *and* exception-on-failure.

## AUTHOR

Damian Conway (damian@conway.org)

## BUGS AND IRRITATIONS

Because it's a module, not an integral part of the interpreter, NEXT.pm has to guess where the surrounding call was found in the method look-up sequence. In the presence of diamond inheritance patterns it occasionally guesses wrong.



It's also too slow (despite caching).

Comment, suggestions, and patches welcome.

**COPYRIGHT**

Copyright (c) 2000–2001, Damian Conway. All Rights Reserved.  
This module is free software. It may be used, redistributed  
and/or modified under the same terms as Perl itself.

**NAME**

Opcode – Disable named opcodes when compiling perl code

**SYNOPSIS**

```
use Opcode;
```

**DESCRIPTION**

Perl code is always compiled into an internal format before execution.

Evaluating perl code (e.g. via "eval" or "do 'file'") causes the code to be compiled into an internal format and then, provided there was no error in the compilation, executed. The internal format is based on many distinct *opcodes*.

By default no opmask is in effect and any code can be compiled.

The Opcode module allow you to define an *operator mask* to be in effect when perl *next* compiles any code. Attempting to compile code which contains a masked opcode will cause the compilation to fail with an error. The code will not be executed.

**NOTE**

The Opcode module is not usually used directly. See the ops pragma and Safe modules for more typical uses.

**WARNING**

The authors make **no warranty**, implied or otherwise, about the suitability of this software for safety or security purposes.

The authors shall not in any case be liable for special, incidental, consequential, indirect or other similar damages arising from the use of this software.

Your mileage will vary. If in any doubt **do not use it**.

**Operator Names and Operator Lists**

The canonical list of operator names is the contents of the array `PL_op_name` defined and initialised in file *opcode.h* of the Perl source distribution (and installed into the perl library).

Each operator has both a terse name (its *opname*) and a more verbose or recognisable descriptive name. The *opdesc* function can be used to return a list of descriptions for a list of operators.

Many of the functions and methods listed below take a list of operators as parameters. Most operator lists can be made up of several types of element. Each element can be one of

an operator name (*opname*)

Operator names are typically small lowercase words like *enterloop*, *leaveloop*, *last*, *next*, *redo* etc. Sometimes they are rather cryptic like *gv2cv*, *i\_ncmp* and *ftsctx*.

an operator tag name (*optag*)

Operator tags can be used to refer to groups (or sets) of operators. Tag names always begin with a colon. The Opcode module defines several *optags* and the user can define others using the *define\_optag* function.

a negated *opname* or *optag*

An *opname* or *optag* can be prefixed with an exclamation mark, e.g., *!mkdir*. Negating an *opname* or *optag* means remove the corresponding ops from the accumulated set of ops at that point.

an operator set (*opset*)

An *opset* as a binary string of approximately 44 bytes which holds a set or zero or more operators.

The `opset` and `opset_to_ops` functions can be used to convert from a list of operators to an opset and *vice versa*.

Wherever a list of operators can be given you can use one or more opsets. See also Manipulating Opsets below.

## Opcode Functions

The Opcode package contains functions for manipulating operator names tags and sets. All are available for export by the package.

**opcodes** In a scalar context `opcodes` returns the number of opcodes in this version of perl (around 350 for perl-5.7.0).

In a list context it returns a list of all the operator names. (Not yet implemented, use `@names = opset_to_ops(full_opset)`.)

**opset (OP, ...)**

Returns an opset containing the listed operators.

**opset\_to\_ops (OPSET)**

Returns a list of operator names corresponding to those operators in the set.

**opset\_to\_hex (OPSET)**

Returns a string representation of an opset. Can be handy for debugging.

**full\_opset** Returns an opset which includes all operators.

**empty\_opset**

Returns an opset which contains no operators.

**invert\_opset (OPSET)**

Returns an opset which is the inverse set of the one supplied.

**verify\_opset (OPSET, ...)**

Returns true if the supplied opset looks like a valid opset (is the right length etc) otherwise it returns false. If an optional second parameter is true then `verify_opset` will croak on an invalid opset instead of returning false.

Most of the other Opcode functions call `verify_opset` automatically and will croak if given an invalid opset.

**define\_optag (OPTAG, OPSET)**

Define OPTAG as a symbolic name for OPSET. Optag names always start with a colon :.

The optag name used must not be defined already (`define_optag` will croak if it is already defined). Optag names are global to the perl process and optag definitions cannot be altered or deleted once defined.

It is strongly recommended that applications using Opcode should use a leading capital letter on their tag names since lowercase names are reserved for use by the Opcode module. If using Opcode within a module you should prefix your tags names with the name of your module to ensure uniqueness and thus avoid clashes with other modules.

**opmask\_add (OPSET)**

Adds the supplied opset to the current opmask. Note that there is currently *no* mechanism for unmasking ops once they have been masked. This is intentional.

**opmask** Returns an opset corresponding to the current opmask.

**opdesc (OP, ...)**

This takes a list of operator names and returns the corresponding list of operator descriptions.

**opdump (PAT)**

Dumps to STDOUT a two column list of op names and op descriptions. If an optional pattern is given then only lines which match the (case insensitive) pattern will be output.

It's designed to be used as a handy command line utility:

```
perl -MOpc=opdump -e opdump
perl -MOpc=opdump -e 'opdump Eval'
```

**Manipulating Opsets**

Opsets may be manipulated using the perl bit vector operators & (and), | (or), ^ (xor) and ~ (negate/invert).

However you should never rely on the numerical position of any opcode within the opset. In other words both sides of a bit vector operator should be opsets returned from Opcode functions.

Also, since the number of opcodes in your current version of perl might not be an exact multiple of eight, there may be unused bits in the last byte of an opset. This should not cause any problems (Opcode functions ignore those extra bits) but it does mean that using the ~ operator will typically not produce the same 'physical' opset 'string' as the invert\_opset function.

**TO DO (maybe)**

```
$bool = opset_eq($opset1, $opset2) true if opsets are logically equiv
$yes = opset_can($opset, @ops) true if $opset has all @ops set
@diff = opset_diff($opset1, $opset2) => ('foo', '!bar', ...)
```

**Predefined Opcode Tags**

:base\_core

```
null stub scalar pushmark wantarray const defined undef
rv2sv sassign
rv2av aassign aelem aelemfast aslice av2arylen
rv2hv helem hslice each values keys exists delete
preinc i_preinc predec i_predec postinc i_postinc postdec i_postdec
int hex oct abs pow multiply i_multiply divide i_divide
modulo i_modulo add i_add subtract i_subtract
left_shift right_shift bit_and bit_xor bit_or negate i_negate
not complement
lt i_lt gt i_gt le i_le ge i_ge eq i_eq ne i_nencmp i_ncmp
slt sgt sle sge seq sne scmp
substr vec stringify study pos length index rindex ord chr
ucfirst lcfirst uc lc quotemeta trans chop schop chomp schomp
match split qr
list lslice splice push pop shift unshift reverse
cond_expr flip flop andassign orassign and or xor
warn die lineseq nextstate scope enter leave setstate
rv2cv anoncode prototype
entersub leavesub leavesublv return method method_named -- XXX loops via rec
leaveeval -- needed for Safe to operate, is safe without entereval
```

**:base\_mem**

These memory related ops are not included in :base\_core because they can easily be used to implement a resource attack (e.g., consume all available memory).

concat repeat join range

anonlist anonhash

Note that despite the existence of this optag a memory resource attack may still be possible using only :base\_core ops.

Disabling these ops is a *very* heavy handed way to attempt to prevent a memory resource attack. It's probable that a specific memory limit mechanism will be added to perl in the near future.

**:base\_loop**

These loop ops are not included in :base\_core because they can easily be used to implement a resource attack (e.g., consume all available CPU time).

grepstart grepwhile  
mapstart mapwhile  
enteriter iter  
enterloop leaveloop unstack  
last next redo  
goto

**:base\_io**

These ops enable *filehandle* (rather than filename) based input and output. These are safe on the assumption that only pre-existing filehandles are available for use. To create new filehandles other ops such as open would need to be enabled.

readline rcatline getc read  
formline enterwrite leavewrite  
print sysread syswrite send recv  
eof tell seek sysseek  
readdir telldir seekdir rewinddir

**:base\_orig**

These are a hotchpotch of opcodes still waiting to be considered

gvsv gv gelem  
padsv padav padhv padany  
rv2gv refgen srefgen ref  
bless -- could be used to change ownership of objects (reblessing)  
pushre regcmaybe regcreset regcomp subst substcont  
sprintf prtf -- can core dump  
crypt  
tie untie  
dbmopen dbmclose  
sselect select  
pipe\_op sockpair  
getppid getpgrp setpgrp getpriority setpriority localtime gmtime

entertry leavetry -- can be used to 'hide' fatal errors

custom -- where should this go

#### :base\_math

These ops are not included in :base\_core because of the risk of them being used to generate floating point exceptions (which would have to be caught using a \$SIG{FPE} handler).

atan2 sin cos exp log sqrt

These ops are not included in :base\_core because they have an effect beyond the scope of the compartment.

rand srand

#### :base\_thread

These ops are related to multi-threading.

lock threadsv

#### :default

A handy tag name for a *reasonable* default set of ops. (The current ops allowed are unstable while development continues. It will change.)

:base\_core :base\_mem :base\_loop :base\_io :base\_orig :base\_thread

If safety matters to you (and why else would you be using the Opcode module?) then you should not rely on the definition of this, or indeed any other, optag!

#### :filesystem\_read

stat lstat readlink

ftatime ftblk ftchr ftctime ftdir fteexec fteowned fteread  
ftewrite ftfile ftis ftlink ftmtime ftpipe ftrexec ftrowned  
ftrread ftsgid ftsize ftsock ftsuid fttty ftzero ftrwrite ftsvtx  
fttext ftbinary

fileno

#### :sys\_db

|                                                   |              |
|---------------------------------------------------|--------------|
| ghbyname ghbyaddr ghostent shostent ehostent      | -- hosts     |
| gnbyname gnbyaddr gnetent snetent enetent         | -- networks  |
| gpbyname gpbynumber gprotoent sprotoent eprotoent | -- protocols |
| gsbyname gsbyport gservent sservent eservent      | -- services  |
| gpwnam gpwuid gpwent spwent epwent getlogin       | -- users     |
| ggrnam ggrgid ggrent sgrent egrent                | -- groups    |

#### :browse

A handy tag name for a *reasonable* default set of ops beyond the :default optag. Like :default (and indeed all the other optags) its current definition is unstable while development continues. It will change.

The :browse tag represents the next step beyond :default. It is a superset of the :default ops and adds :filesystem\_read the :sys\_db. The intent being that scripts can access more (possibly sensitive) information about your system but not be able to change it.

:default :filesystem\_read :sys\_db

#### :filesystem\_open

sysopen open close  
umask binmode

```

 open_dir closedir -- other dir ops are in :base_io
:fileys_write
 link unlink rename symlink truncate
 mkdir rmdir
 utime chmod chown
 fcntl -- not strictly fileys related, but possibly as dangerous?
:subprocess
 backtick system
 fork
 wait waitpid
 glob -- access to Cshell via <'rm *'>
:ownprocess
 exec exit kill
 time tms -- could be used for timing attacks (paranoid?)
:others
 This tag holds groups of assorted specialist opcodes that don't warrant having optags defined for
 them.
 SystemV Interprocess Communications:
 msgctl msgget msgrcv msgsnd
 semctl semget semop
 shmctl shmget shmread shmwrite
:still_to_be_decided
 chdir
 flock ioctl
 socket getpeername sockopt
 bind connect listen accept shutdown gsockopt getsockname
 sleep alarm -- changes global timer state and signal handling
 sort -- assorted problems including core dumps
 tied -- can be used to access object implementing a tie
 pack unpack -- can be used to create/use memory pointers
 entereval -- can be used to hide code from initial compile
 require dofile
 caller -- get info about calling environment and args
 reset
 dbstate -- perl -d version of nextstate(ment) opcode
:dangerous
 This tag is simply a bucket for opcodes that are unlikely to be used via a tag name but need to be
 tagged for completeness and documentation.
 syscall dump chroot

```

**SEE ALSO**

ops(3) — perl pragma interface to Opcode module.

Safe(3) — Opcode and namespace limited execution compartments

**AUTHORS**

Originally designed and implemented by Malcolm Beattie, mbeattie@sable.ox.ac.uk as part of Safe version 1.

Split out from Safe module version 1, named opcode tags and other changes added by Tim Bunce.



**NAME**

ops – Perl pragma to restrict unsafe operations when compiling

**SYNOPSIS**

```
perl -Mops=:default ... # only allow reasonably safe operations
perl -M-ops=system ... # disable the 'system' opcode
```

**DESCRIPTION**

Since the ops pragma currently has an irreversible global effect, it is only of significant practical use with the `-M` option on the command line.

See the [Opcode](#) module for information about opcodes, optags, opmasks and important information about safety.

**SEE ALSO**

Opcode(3), Safe(3), perlrun(3)

## NAME

Safe – Compile and execute code in restricted compartments

## SYNOPSIS

```
use Safe;

$compartment = new Safe;

$compartment->permit(qw(time sort :browse));

$result = $compartment->reval($unsafe_code);
```

## DESCRIPTION

The Safe extension module allows the creation of compartments in which perl code can be evaluated. Each compartment has

### a new namespace

The "root" of the namespace (i.e. "main::") is changed to a different package and code evaluated in the compartment cannot refer to variables outside this namespace, even with run-time glob lookups and other tricks.

Code which is compiled outside the compartment can choose to place variables into (or *share* variables with) the compartment's namespace and only that data will be visible to code evaluated in the compartment.

By default, the only variables shared with compartments are the "underscore" variables `$_` and `@_` (and, technically, the less frequently used `%_`, the `_` filehandle and so on). This is because otherwise perl operators which default to `$_` will not work and neither will the assignment of arguments to `@_` on subroutine entry.

### an operator mask

Each compartment has an associated "operator mask". Recall that perl code is compiled into an internal format before execution. Evaluating perl code (e.g. via "eval" or "do 'file'") causes the code to be compiled into an internal format and then, provided there was no error in the compilation, executed. Code evaluated in a compartment compiles subject to the compartment's operator mask. Attempting to evaluate code in a compartment which contains a masked operator will cause the compilation to fail with an error. The code will not be executed.

The default operator mask for a newly created compartment is the `':default'` optag.

It is important that you read the Opcode(3) module documentation for more information, especially for detailed definitions of opnames, optags and opsets.

Since it is only at the compilation stage that the operator mask applies, controlled access to potentially unsafe operations can be achieved by having a handle to a wrapper subroutine (written outside the compartment) placed into the compartment. For example,

```
$cpt = new Safe;
sub wrapper {
 # vet arguments and perform potentially unsafe operations
}
$cpt->share('&wrapper');
```

## WARNING

The authors make **no warranty**, implied or otherwise, about the suitability of this software for safety or security purposes.

The authors shall not in any case be liable for special, incidental, consequential, indirect or other similar damages arising from the use of this software.

Your mileage will vary. If in any doubt **do not use it**.

## RECENT CHANGES

The interface to the Safe module has changed quite dramatically since version 1 (as supplied with Perl5.002). Study these pages carefully if you have code written to use Safe version 1 because you will need to make changes.

## Methods in class Safe

To create a new compartment, use

```
$cpt = new Safe;
```

Optional argument is (NAMESPACE), where NAMESPACE is the root namespace to use for the compartment (defaults to "Safe::Root0", incremented for each new compartment).

Note that version 1.00 of the Safe module supported a second optional parameter, MASK. That functionality has been withdrawn pending deeper consideration. Use the permit and deny methods described below.

The following methods can then be used on the compartment object returned by the above constructor. The object argument is implicit in each case.

permit (OP, ...)

Permit the listed operators to be used when compiling code in the compartment (in *addition* to any operators already permitted).

permit\_only (OP, ...)

Permit *only* the listed operators to be used when compiling code in the compartment (*no* other operators are permitted).

deny (OP, ...)

Deny the listed operators from being used when compiling code in the compartment (other operators may still be permitted).

deny\_only (OP, ...)

Deny *only* the listed operators from being used when compiling code in the compartment (*all* other operators will be permitted).

trap (OP, ...)

untrap (OP, ...)

The trap and untrap methods are synonyms for deny and permit respectively.

share (NAME, ...)

This shares the variable(s) in the argument list with the compartment. This is almost identical to exporting variables using the [Exporter](#) module.

Each NAME must be the **name** of a variable, typically with the leading type identifier included. A bareword is treated as a function name.

Examples of legal names are '\$foo' for a scalar, '@foo' for an array, '%foo' for a hash, '&foo' or 'foo' for a subroutine and '\*foo' for a glob (i.e. all symbol table entries associated with "foo", including scalar, array, hash, sub and filehandle).

Each NAME is assumed to be in the calling package. See share\_from for an alternative method (which share uses).

share\_from (PACKAGE, ARRAYREF)

This method is similar to share () but allows you to explicitly name the package that symbols should be shared from. The symbol names (including type characters) are supplied as an array reference.

```
$safe->share_from('main', ['$foo', '%bar', 'func']);
```

**varglob (VARNAME)**

This returns a glob reference for the symbol table entry of VARNAME in the package of the compartment. VARNAME must be the **name** of a variable without any leading type marker. For example,

```
$cpt = new Safe 'Root';
$Root::foo = "Hello world";
Equivalent version which doesn't need to know $cpt's package name:
${$cpt->varglob('foo')} = "Hello world";
```

**reval (STRING)**

This evaluates STRING as perl code inside the compartment.

The code can only see the compartment's namespace (as returned by the **root** method). The compartment's root package appears to be the `main::` package to the code inside the compartment.

Any attempt by the code in STRING to use an operator which is not permitted by the compartment will cause an error (at run-time of the main program but at compile-time for the code in STRING). The error is of the form "%s trapped by operation mask operation...".

If an operation is trapped in this way, then the code in STRING will not be executed. If such a trapped operation occurs or any other compile-time or return error, then `$@` is set to the error message, just as with an `eval()`.

If there is no error, then the method returns the value of the last expression evaluated, or a return statement may be used, just as with subroutines and **eval()**. The context (list or scalar) is determined by the caller as usual.

This behaviour differs from the beta distribution of the Safe extension where earlier versions of perl made it hard to mimic the return behaviour of the `eval()` command and the context was always scalar.

Some points to note:

If the `entereval` op is permitted then the code can use `eval "..."` to 'hide' code which might use denied ops. This is not a major problem since when the code tries to execute the `eval` it will fail because the `opmask` is still in effect. However this technique would allow clever, and possibly harmful, code to 'probe' the boundaries of what is possible.

Any string eval which is executed by code executing in a compartment, or by code called from code executing in a compartment, will be eval'd in the namespace of the compartment. This is potentially a serious problem.

Consider a function `foo()` in package `pkg` compiled outside a compartment but shared with it. Assume the compartment has a root package called 'Root'. If `foo()` contains an `eval` statement like `eval '$foo = 1'` then, normally, `$pkg::foo` will be set to 1. If `foo()` is called from the compartment (by whatever means) then instead of setting `$pkg::foo`, the eval will actually set `$Root::pkg::foo`.

This can easily be demonstrated by using a module, such as the `Socket` module, which uses `eval "..."` as part of an `AUTOLOAD` function. You can 'use' the module outside the compartment and share an (autoloaded) function with the compartment. If an autoload is triggered by code in the compartment, or by any code anywhere that is called by any means from the compartment, then the `eval` in the `Socket` module's `AUTOLOAD` function happens in the namespace of the compartment. Any variables created or used by the eval'd code are now under the control of the code in the compartment.

A similar effect applies to *all* runtime symbol lookups in code called from a compartment but not compiled within it.

**rdo (FILENAME)**

This evaluates the contents of file FILENAME inside the compartment. See above documentation on the **reval** method for further details.

**root (NAMESPACE)**

This method returns the name of the package that is the root of the compartment's namespace.

Note that this behaviour differs from version 1.00 of the Safe module where the root module could be used to change the namespace. That functionality has been withdrawn pending deeper consideration.

**mask (MASK)**

This is a get-or-set method for the compartment's operator mask.

With no MASK argument present, it returns the current operator mask of the compartment.

With the MASK argument present, it sets the operator mask for the compartment (equivalent to calling the deny\_only method).

**Some Safety Issues**

This section is currently just an outline of some of the things code in a compartment might do (intentionally or unintentionally) which can have an effect outside the compartment.

**Memory** Consuming all (or nearly all) available memory.

**CPU** Causing infinite loops etc.

**Snooping** Copying private information out of your system. Even something as simple as your user name is of value to others. Much useful information could be gleaned from your environment variables for example.

**Signals** Causing signals (especially SIGFPE and SIGALARM) to affect your process.

Setting up a signal handler will need to be carefully considered and controlled. What mask is in effect when a signal handler gets called? If a user can get an imported function to get an exception and call the user's signal handler, does that user's restricted mask get re-instated before the handler is called? Does an imported handler get called with its original mask or the user's one?

**State Changes**

Ops such as chdir obviously effect the process as a whole and not just the code in the compartment. Ops such as rand and srand have a similar but more subtle effect.

**AUTHOR**

Originally designed and implemented by Malcolm Beattie, mbeattie@sable.ox.ac.uk.

Reworked to use the Opcode module and other changes added by Tim Bunce <Tim.Bunce@ig.co.uk>.

**NAME**

open – perl pragma to set default PerlIO layers for input and output

**SYNOPSIS**

```
use open IN => ":crlf", OUT => ":bytes";
use open OUT => ':utf8';
use open IO => ":encoding(iso-8859-7)";

use open IO => ':locale';

use open ':utf8';
use open ':locale';
use open ':encoding(iso-8859-7)';

use open ':std';
```

**DESCRIPTION**

Full-fledged support for I/O layers is now implemented provided Perl is configured to use PerlIO as its IO system (which is now the default).

The open pragma serves as one of the interfaces to declare default "layers" (also known as "disciplines") for all I/O. Any open(), readpipe() (aka qx//) and similar operators found within the lexical scope of this pragma will use the declared defaults.

With the IN subpragma you can declare the default layers of input streams, and with the OUT subpragma you can declare the default layers of output streams. With the IO subpragma you can control both input and output streams simultaneously.

If you have a legacy encoding, you can use the :encoding(...) tag.

if you want to set your encoding layers based on your locale environment variables, you can use the :locale tag. For example:

```
$ENV{LANG} = 'ru_RU.KOI8-R';
the :locale will probe the locale environment variables like LANG
use open OUT => ':locale';
open(O, ">koi8");
print O chr(0x430); # Unicode CYRILLIC SMALL LETTER A = KOI8-R 0xc1
close O;
open(I, "<koi8");
printf "%#x\n", ord(<I>), "\n"; # this should print 0xc1
close I;
```

These are equivalent

```
use open ':utf8';
use open IO => ':utf8';
```

as are these

```
use open ':locale';
use open IO => ':locale';
```

and these

```
use open ':encoding(iso-8859-7)';
use open IO => ':encoding(iso-8859-7)';
```

The matching of encoding names is loose: case does not matter, and many encodings have several aliases. See [Encode::Supported](#) for details and the list of supported locales.

Note that :utf8 PerlIO layer must always be specified exactly like that, it is not subject to the loose

matching of encoding names.

When `open()` is given an explicit list of layers they are appended to the list declared using this pragma.

The `:std` subpragma on its own has no effect, but if combined with the `:utf8` or `:encoding` subpragmas, it converts the standard filehandles (STDIN, STDOUT, STDERR) to comply with encoding selected for input/output handles. For example, if both input and out are chosen to be `:utf8`, a `:std` will mean that STDIN, STDOUT, and STDERR are also in `:utf8`. On the other hand, if only output is chosen to be in `< :encoding(koi8r)`, a `:std` will cause only the STDOUT and STDERR to be in `koi8r`. The `:locale` subpragma implicitly turns on `:std`.

The logic of `:locale` is as follows:

1. If the platform supports the `langinfo(CODESET)` interface, the codeset returned is used as the default encoding for the open pragma.
2. If 1. didn't work but we are under the locale pragma, the environment variables `LC_ALL` and `LANG` (in that order) are matched for encodings (the part after `.`, if any), and if any found, that is used as the default encoding for the open pragma.
3. If 1. and 2. didn't work, the environment variables `LC_ALL` and `LANG` (in that order) are matched for anything looking like UTF-8, and if any found, `:utf8` is used as the default encoding for the open pragma.

If your locale environment variables (`LANGUAGE`, `LC_ALL`, `LC_CTYPE`, `LANG`) contain the strings 'UTF-8' or 'UTF8' (case-insensitive matching), the default encoding of your STDIN, STDOUT, and STDERR, and of **any subsequent file open**, is UTF-8.

Directory handles may also support PerlIO layers in the future.

## NONPERLIO FUNCTIONALITY

If Perl is not built to use PerlIO as its IO system then only the two pseudo-layers `:bytes` and `:crlf` are available.

The `:bytes` layer corresponds to "binary mode" and the `:crlf` layer corresponds to "text mode" on platforms that distinguish between the two modes when opening files (which is many DOS-like platforms, including Windows). These two layers are no-ops on platforms where `binmode()` is a no-op, but perform their functions everywhere if PerlIO is enabled.

## IMPLEMENTATION DETAILS

There is a class method in `PerlIO::Layer` `find` which is implemented as XS code. It is called by `import` to validate the layers:

```
PerlIO::Layer::->find("perlio")
```

The return value (if defined) is a Perl object, of class `PerlIO::Layer` which is created by the C code in *perlio.c*. As yet there is nothing useful you can do with the object at the perl level.

## SEE ALSO

*binmode in perlfunc*, *open in perlfunc*, *perlunicode*, *PerlIO*, *encoding*

**NAME**

overload – Package for overloading perl operations

**SYNOPSIS**

```
package Something;

use overload
 '+' => \&myadd,
 '-' => \&mysub;
 # etc
...

package main;
$a = new Something 57;
$b=5+$a;
...
if (overload::Overloaded $b) {...}
...
$strval = overload::StrVal $b;
```

**DESCRIPTION****Declaration of overloaded functions**

The compilation directive

```
package Number;
use overload
 "+" => \&add,
 "*" => "muas";
```

declares function `Number::add()` for addition, and method `muas()` in the "class" `Number` (or one of its base classes) for the assignment form `*=` of multiplication.

Arguments of this directive come in (key, value) pairs. Legal values are values legal inside a `&{ ... }` call, so the name of a subroutine, a reference to a subroutine, or an anonymous subroutine will all work. Note that values specified as strings are interpreted as methods, not subroutines. Legal keys are listed below.

The subroutine `add` will be called to execute `$a+$b` if `$a` is a reference to an object blessed into the package `Number`, or if `$a` is not an object from a package with defined mathematic addition, but `$b` is a reference to a `Number`. It can also be called in other situations, like `$a+=7`, or `$a++`. See [MAGIC AUTOGENERATION](#). (Mathematical methods refer to methods triggered by an overloaded mathematical operator.)

Since overloading respects inheritance via the `@ISA` hierarchy, the above declaration would also trigger overloading of `+` and `*=` in all the packages which inherit from `Number`.

**Calling Conventions for Binary Operations**

The functions specified in the `use overload ...` directive are called with three (in one particular case with four, see [Last Resort](#)) arguments. If the corresponding operation is binary, then the first two arguments are the two arguments of the operation. However, due to general object calling conventions, the first argument should always be an object in the package, so in the situation of `7+$a`, the order of the arguments is interchanged. It probably does not matter when implementing the addition method, but whether the arguments are reversed is vital to the subtraction method. The method can query this information by examining the third argument, which can take three different values:

**FALSE** the order of arguments is as in the current operation.



**TRUE** the arguments are reversed.

**undef** the current operation is an assignment variant (as in `$a+=7`), but the usual function is called instead. This additional information can be used to generate some optimizations. Compare [Calling Conventions for Mutators](#).

### Calling Conventions for Unary Operations

Unary operation are considered binary operations with the second argument being `undef`. Thus the functions that overloads `{ "++" }` is called with arguments `( $a, undef, ' ' )` when `$a++` is executed.

### Calling Conventions for Mutators

Two types of mutators have different calling conventions:

**++ and --**

The routines which implement these operators are expected to actually *mutate* their arguments. So, assuming that `$obj` is a reference to a number,

```
sub incr { my $n = $ {$_[0]}; ++$n; $_[0] = bless \$n }
```

is an appropriate implementation of overloaded `++`. Note that

```
sub incr { ++$ {$_[0]} ; shift }
```

is OK if used with preincrement and with postincrement. (In the case of postincrement a copying will be performed, see [Copy Constructor](#).)

**x= and other assignment versions**

There is nothing special about these methods. They may change the value of their arguments, and may leave it as is. The result is going to be assigned to the value in the left-hand-side if different from this value.

This allows for the same method to be used as overloaded `+=` and `+`. Note that this is *allowed*, but not recommended, since by the semantic of *"Fallback"* Perl will call the method for `+` anyway, if `+=` is not overloaded.

**Warning.** Due to the presence of assignment versions of operations, routines which may be called in assignment context may create self-referential structures. Currently Perl will not free self-referential structures until cycles are explicitly broken. You may get problems when traversing your structures too.

Say,

```
use overload '+' => sub { bless [$_[0], $_[1]] };
```

is asking for trouble, since for code `$obj += $foo` the subroutine is called as `$obj = add($obj, $foo, undef)`, or `$obj = [ $_[0], $_[1] ]`. If using such a subroutine is an important optimization, one can overload `+=` explicitly by a non-"optimized" version, or switch to non-optimized version if not defined `$_[2]` (see [Calling Conventions for Binary Operations](#)).

Even if no *explicit* assignment-variants of operators are present in the script, they may be generated by the optimizer. Say, `"$, $obj, "` or `' , ' . $obj . ' , ' .` may be both optimized to

```
my $tmp = ' , ' . $obj; $tmp .= ' , ' ;
```

### Overloadable Operations

The following symbols can be specified in use `overload` directive:

- *Arithmetic operations*

```
"+", "+=", "-", "-=", "*", "*=", "/", "/=", "%", "%=",
"**", "**=", "<<", "<<=", ">>", ">>=", "x", "x=", ".", ".=",
```

For these operations a substituted non-assignment variant can be called if the assignment variant is

not available. Methods for operations `+`, `-`, `+=`, and `-=` can be called to automatically generate increment and decrement methods. The operation `-` can be used to autogenerate missing methods for unary minus or `abs`.

See "[MAGIC AUTOGENERATION](#)", "[Calling Conventions for Mutators](#)" and "[Calling Conventions for Binary Operations](#)") for details of these substitutions.

- *Comparison operations*

```
"<", "<=", ">", ">=", "==", "!=", "<=>",
"lt", "le", "gt", "ge", "eq", "ne", "cmp",
```

If the corresponding "spaceship" variant is available, it can be used to substitute for the missing operation. During sorting arrays, `cmp` is used to compare values subject to use `overload`.

- *Bit operations*

```
"&", "^", "|", "neg", "!", "~",
```

`neg` stands for unary minus. If the method for `neg` is not specified, it can be autogenerated using the method for subtraction. If the method for `!` is not specified, it can be autogenerated using the methods for `bool`, or `"`, or `0+`.

- *Increment and decrement*

```
"++", "--",
```

If undefined, addition and subtraction methods can be used instead. These operations are called both in prefix and postfix form.

- *Transcendental functions*

```
"atan2", "cos", "sin", "exp", "abs", "log", "sqrt", "int"
```

If `abs` is unavailable, it can be autogenerated using methods for `"<"` or `"<="` combined with either unary minus or subtraction.

Note that traditionally the Perl function `int` rounds to 0, thus for floating-point-like types one should follow the same semantic. If `int` is unavailable, it can be autogenerated using the overloading of `0+`.

- *Boolean, string and numeric conversion*

```
'bool', '""', '0+',
```

If one or two of these operations are not overloaded, the remaining ones can be used instead. `bool` is used in the flow control operators (like `while`) and for the ternary `?:` operation. These functions can return any arbitrary Perl value. If the corresponding operation for this value is overloaded too, that operation will be called again with this value.

As a special case if the overload returns the object itself then it will be used directly. An overloaded conversion returning the object is probably a bug, because you're likely to get something that looks like `YourPackage=HASH(0x8172b34)`.

- *Iteration*

```
"<>"
```

If not overloaded, the argument will be converted to a filehandle or glob (which may require a stringification). The same overloading happens both for the *read-filehandle* syntax `<$var>` and *globbing* syntax `<${var}>`.

**BUGS** Even in list context, the iterator is currently called only once and with scalar context.

- *Dereferencing*

```
'${}', '@{%', '%{%', '&{%', '*{%'.
```

If not overloaded, the argument will be dereferenced *as is*, thus should be of correct type. These

functions should return a reference of correct type, or another object with overloaded dereferencing.

As a special case if the overload returns the object itself then it will be used directly (provided it is the correct type).

The dereference operators must be specified explicitly they will not be passed to "nomethod".

- *Special*

"nomethod", "fallback", "=",

see [SPECIAL SYMBOLS FOR use overload](#).

See ["Fallback"](#) for an explanation of when a missing method can be autogenerated.

A computer-readable form of the above table is available in the hash %overload::ops, with values being space-separated lists of names:

```
with_assign => '+ - * / % ** << >> x .',
assign => '+= -= *= /= %= **= <=> >>= x= .=',
num_comparison => '< <= > >= == !=',
'3way_comparison'=> '<=> cmp',
str_comparison => 'lt le gt ge eq ne',
binary => '& | ^',
unary => 'neg ! ~',
mutators => '++ --',
func => 'atan2 cos sin exp abs log sqrt',
conversion => 'bool "" 0+',
iterators => '<>',
dereferencing => '${} @{} %{} &{} *{}',
special => 'nomethod fallback ='
```

## Inheritance and overloading

Inheritance interacts with overloading in two ways.

### Strings as values of use overload directive

If value in

```
use overload key => value;
```

is a string, it is interpreted as a method name.

### Overloading of an operation is inherited by derived classes

Any class derived from an overloaded class is also overloaded. The set of overloaded methods is the union of overloaded methods of all the ancestors. If some method is overloaded in several ancestor, then which description will be used is decided by the usual inheritance rules:

If A inherits from B and C (in this order), B overloads + with \&D::plus\_sub, and C overloads + by "plus\_meth", then the subroutine D::plus\_sub will be called to implement operation + for an object in package A.

Note that since the value of the fallback key is not a subroutine, its inheritance is not governed by the above rules. In the current implementation, the value of fallback in the first overloaded ancestor is used, but this is accidental and subject to change.

## SPECIAL SYMBOLS FOR use overload

Three keys are recognized by Perl that are not covered by the above description.

### Last Resort

"nomethod" should be followed by a reference to a function of four parameters. If defined, it is called when the overloading mechanism cannot find a method for some operation. The first three arguments of this function coincide with the arguments for the corresponding method if it were found, the fourth argument is

the symbol corresponding to the missing method. If several methods are tried, the last one is used. Say, `1-$a` can be equivalent to

```
&nomethodMethod($a, 1, 1, "-")
```

if the pair `"nomethod" => "nomethodMethod"` was specified in the `use overload` directive.

The `"nomethod"` mechanism is *not* used for the dereference operators (`{}`, `@{}`, `%{}` & `{}` `*{}`).

If some operation cannot be resolved, and there is no function assigned to `"nomethod"`, then an exception will be raised via `die()` — unless `"fallback"` was specified as a key in `use overload` directive.

### Fallback

The key `"fallback"` governs what to do if a method for a particular operation is not found. Three different cases are possible depending on the value of `"fallback"`:

- `undef` Perl tries to use a substituted method (see [MAGIC AUTOGENERATION](#)). If this fails, it then tries to call `"nomethod"` value; if missing, an exception will be raised.
- `TRUE` The same as for the `undef` value, but no exception is raised. Instead, it silently reverts to what it would have done were there no `use overload` present.
- `defined, but FALSE` No autogeneration is tried. Perl tries to call `"nomethod"` value, and if this is missing, raises an exception.

**Note.** `"fallback"` inheritance via `@ISA` is not carved in stone yet, see ["Inheritance and overloading"](#).

### Copy Constructor

The value for `"="` is a reference to a function with three arguments, i.e., it looks like the other values in `use overload`. However, it does not overload the Perl assignment operator. This would go against Camel hair.

This operation is called in the situations when a mutator is applied to a reference that shares its object with some other reference, such as

```
$a=$b;
++$a;
```

To make this change `$a` and not change `$b`, a copy of `$$a` is made, and `$a` is assigned a reference to this new object. This operation is done during execution of the `++$a`, and not during the assignment, (so before the increment `$$a` coincides with `$$b`). This is only done if `++` is expressed via a method for `'++'` or `'+='` (or `nomethod`). Note that if this operation is expressed via `'+'` a nonmutator, i.e., as in

```
$a=$b;
$a=$a+1;
```

then `$a` does not reference a new copy of `$$a`, since `$$a` does not appear as lvalue when the above code is executed.

If the copy constructor is required during the execution of some mutator, but a method for `'='` was not specified, it can be autogenerated as a string copy if the object is a plain scalar.

### Example

The actually executed code for

```
$a=$b;
Something else which does not modify $a or $b....
++$a;
```

may be

```
$a=$b;
Something else which does not modify $a or $b....
$a = $a->clone(undef,"");
```

```
$a->incr(undef, "");
```

if \$b was mathematical, and '++' was overloaded with \&incr, '=' was overloaded with \&clone.

Same behaviour is triggered by \$b = \$a++, which is consider a synonym for \$b = \$a; ++\$a.

## MAGIC AUTOGENERATION

If a method for an operation is not found, and the value for "fallback" is TRUE or undefined, Perl tries to autogenerate a substitute method for the missing operation based on the defined operations. Autogenerated method substitutions are possible for the following operations:

### *Assignment forms of arithmetic operations*

\$a+=\$b can use the method for "+" if the method for "+=" is not defined.

### *Conversion operations*

String, numeric, and boolean conversion are calculated in terms of one another if not all of them are defined.

### *Increment and decrement*

The ++\$a operation can be expressed in terms of \$a+=1 or \$a+1, and \$a- in terms of \$a-=1 and \$a-1.

abs(\$a) can be expressed in terms of \$a<0 and -\$a (or 0-\$a).

*Unary minus* can be expressed in terms of subtraction.

*Negation* ! and not can be expressed in terms of boolean conversion, or string or numerical conversion.

*Concatenation* can be expressed in terms of string conversion.

### *Comparison operations*

can be expressed in terms of its "spaceship" counterpart: either <=> or cmp:

|                        |                 |
|------------------------|-----------------|
| <, >, <=, >=, ==, !=   | in terms of <=> |
| lt, gt, le, ge, eq, ne | in terms of cmp |

### *Iterator*

<> in terms of builtin operations

### *Dereferencing*

\${} @{} %{} &{} \*{} in terms of builtin operations

*Copy operator* can be expressed in terms of an assignment to the dereferenced value, if this value is a scalar and not a reference.

## Losing overloading

The restriction for the comparison operation is that even if, for example, 'cmp' should return a blessed reference, the autogenerated 'lt' function will produce only a standard logical value based on the numerical value of the result of 'cmp'. In particular, a working numeric conversion is needed in this case (possibly expressed in terms of other conversions).

Similarly, .= and x= operators lose their mathematical properties if the string conversion substitution is applied.

When you chop() a mathematical object it is promoted to a string and its mathematical properties are lost. The same can happen with other operations as well.

## Run-time Overloading

Since all use directives are executed at compile-time, the only way to change overloading during run-time is to

```
eval 'use overload "+" => \&addmethod';
```

You can also use

```
eval 'no overload "+", "--", "<="';
```

though the use of these constructs during run-time is questionable.

## Public functions

Package `overload.pm` provides the following public functions:

`overload::StrVal(arg)`

Gives string value of `arg` as in absence of stringify overloading.

`overload::Overloaded(arg)`

Returns true if `arg` is subject to overloading of some operations.

`overload::Method(obj,op)`

Returns `undef` or a reference to the method that implements `op`.

## Overloading constants

For some application Perl parser mangles constants too much. It is possible to hook into this process via `overload::constant()` and `overload::remove_constant()` functions.

These functions take a hash as an argument. The recognized keys of this hash are

|                      |                                                                                                                                          |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <code>integer</code> | to overload integer constants,                                                                                                           |
| <code>float</code>   | to overload floating point constants,                                                                                                    |
| <code>binary</code>  | to overload octal and hexadecimal constants,                                                                                             |
| <code>q</code>       | to overload <code>q</code> -quoted strings, constant pieces of <code>qq</code> - and <code>qx</code> -quoted strings and here-documents, |
| <code>qr</code>      | to overload constant pieces of regular expressions.                                                                                      |

The corresponding values are references to functions which take three arguments: the first one is the *initial* string form of the constant, the second one is how Perl interprets this constant, the third one is how the constant is used. Note that the initial string form does not contain string delimiters, and has backslashes in backslash-delimiter combinations stripped (thus the value of delimiter is not relevant for processing of this string). The return value of this function is how this constant is going to be interpreted by Perl. The third argument is undefined unless for overloaded `q`- and `qr`- constants, it is `q` in single-quote context (comes from strings, regular expressions, and single-quote HERE documents), it is `tr` for arguments of `tr/y` operators, it is `s` for right-hand side of `s`-operator, and it is `qq` otherwise.

Since an expression `"ab$cd,,"` is just a shortcut for `'ab' . $cd . ',,''`, it is expected that overloaded constant strings are equipped with reasonable overloaded catenation operator, otherwise absurd results will result. Similarly, negative numbers are considered as negations of positive constants.

Note that it is probably meaningless to call the functions `overload::constant()` and `overload::remove_constant()` from anywhere but `import()` and `unimport()` methods. From these methods they may be called as

```
sub import {
 shift;
 return unless @_;
 die "unknown import: @_ " unless @_ == 1 and $_[0] eq ':constant';
```

```

 overload::constant integer => sub {Math::BigInt->new(shift)};
 }

```

**BUGS** Currently overloaded-ness of constants does not propagate into `eval '...'`.

## IMPLEMENTATION

What follows is subject to change RSN.

The table of methods for all operations is cached in magic for the symbol table hash for the package. The cache is invalidated during processing of `use overload`, `no overload`, new function definitions, and changes in `@ISA`. However, this invalidation remains unprocessed until the next `blessing` into the package. Hence if you want to change overloading structure dynamically, you'll need an additional (fake) `blessing` to update the table.

(Every SVish thing has a magic queue, and magic is an entry in that queue. This is how a single variable may participate in multiple forms of magic simultaneously. For instance, environment variables regularly have two forms at once: their `%ENV` magic and their taint magic. However, the magic which implements overloading is applied to the stashes, which are rarely used directly, thus should not slow down Perl.)

If an object belongs to a package using `overload`, it carries a special flag. Thus the only speed penalty during arithmetic operations without overloading is the checking of this flag.

In fact, if `use overload` is not present, there is almost no overhead for overloadable operations, so most programs should not suffer measurable performance penalties. A considerable effort was made to minimize the overhead when `overload` is used in some package, but the arguments in question do not belong to packages using `overload`. When in doubt, test your speed with `use overload` and without it. So far there have been no reports of substantial speed degradation if Perl is compiled with optimization turned on.

There is no size penalty for data if `overload` is not used. The only size penalty if `overload` is used in some package is that *all* the packages acquire a magic during the next `blessing` into the package. This magic is three-words-long for packages without overloading, and carries the cache table if the package is overloaded.

Copying (`$a=$b`) is shallow; however, a one-level-deep copying is carried out before any operation that can imply an assignment to the object `$a` (or `$b`) refers to, like `$a++`. You can override this behavior by defining your own copy constructor (see "[Copy Constructor](#)").

It is expected that arguments to methods that are not explicitly supposed to be changed are constant (but this is not enforced).

## Metaphor clash

One may wonder why the semantic of overloaded `=` is so counter intuitive. If it *looks* counter intuitive to you, you are subject to a metaphor clash.

Here is a Perl object metaphor:

*object is a reference to blessed data*

and an arithmetic metaphor:

*object is a thing by itself.*

The *main* problem of overloading `=` is the fact that these metaphors imply different actions on the assignment `$a = $b` if `$a` and `$b` are objects. Perl-think implies that `$a` becomes a reference to whatever `$b` was referencing. Arithmetic-think implies that the value of "object" `$a` is changed to become the value of the object `$b`, preserving the fact that `$a` and `$b` are separate entities.

The difference is not relevant in the absence of mutators. After a Perl-way assignment an operation which mutates the data referenced by `$a` would change the data referenced by `$b` too. Effectively, after `$a = $b` values of `$a` and `$b` become *indistinguishable*.

On the other hand, anyone who has used algebraic notation knows the expressive power of the arithmetic metaphor. Overloading works hard to enable this metaphor while preserving the Perlian way as far as

possible. Since it is not possible to freely mix two contradicting metaphors, overloading allows the arithmetic way to write things *as far as all the mutators are called via overloaded access only*. The way it is done is described in [Copy Constructor](#).

If some mutator methods are directly applied to the overloaded values, one may need to *explicitly unlink* other values which references the same value:

```
$a = new Data 23;
...
$b = $a; # $b is "linked" to $a
...
$a = $a->clone; # Unlink $b from $a
$a->increment_by(4);
```

Note that overloaded access makes this transparent:

```
$a = new Data 23;
$b = $a; # $b is "linked" to $a
$a += 4; # would unlink $b automatically
```

However, it would not make

```
$a = new Data 23;
$a = 4; # Now $a is a plain 4, not 'Data'
```

preserve "objectness" of `$a`. But Perl *has* a way to make assignments to an object do whatever you want. It is just not the overload, but `tie()`ing interface (see [tie](#)). Adding a `FETCH()` method which returns the object itself, and `STORE()` method which changes the value of the object, one can reproduce the arithmetic metaphor in its completeness, at least for variables which were `tie()`d from the start.

(Note that a workaround for a bug may be needed, see ["BUGS"](#).)

## Cookbook

Please add examples to what follows!

### Two-face scalars

Put this in *two\_face.pm* in your Perl library directory:

```
package two_face; # Scalars with separate string and
 # numeric values.
sub new { my $p = shift; bless [@_], $p }
use overload '""' => \&str, '0+' => \&num, fallback => 1;
sub num {shift->[1]}
sub str {shift->[0]}
```

Use it as follows:

```
require two_face;
my $seven = new two_face ("vii", 7);
printf "seven=$seven, seven=%d, eight=%d\n", $seven, $seven+1;
print "seven contains 'i'\n" if $seven =~ /i/;
```

(The second line creates a scalar which has both a string value, and a numeric value.) This prints:

```
seven=vii, seven=7, eight=8
seven contains 'i'
```

### Two-face references

Suppose you want to create an object which is accessible as both an array reference and a hash reference, similar to the [Pseudo-hashes: Using an array as a hash in pseudo-hash/perlref](#) builtin Perl type. Let's make it better than a pseudo-hash by allowing index 0 to be treated as a normal element.



```

package two_refs;
use overload '%{}' => \&gethash, '@{}' => sub { $ {shift()} };
sub new {
 my $p = shift;
 bless \ [@_], $p;
}
sub gethash {
 my %h;
 my $self = shift;
 tie %h, ref $self, $self;
 \%h;
}

sub TIEHASH { my $p = shift; bless \ shift, $p }
my %fields;
my $i = 0;
$fields{$_} = $i++ foreach qw{zero one two three};
sub STORE {
 my $self = ${shift()};
 my $key = $fields{shift()};
 defined $key or die "Out of band access";
 $$self->[$key] = shift;
}
sub FETCH {
 my $self = ${shift()};
 my $key = $fields{shift()};
 defined $key or die "Out of band access";
 $$self->[$key];
}

```

Now one can access an object using both the array and hash syntax:

```

my $bar = new two_refs 3,4,5,6;
$bar->[2] = 11;
$bar->{two} == 11 or die 'bad hash fetch';

```

Note several important features of this example. First of all, the *actual* type of `$bar` is a scalar reference, and we do not overload the scalar dereference. Thus we can get the *actual* non-overloaded contents of `$bar` by just using `$$bar` (what we do in functions which overload dereference). Similarly, the object returned by the `TIEHASH()` method is a scalar reference.

Second, we create a new tied hash each time the hash syntax is used. This allows us not to worry about a possibility of a reference loop, which would lead to a memory leak.

Both these problems can be cured. Say, if we want to overload hash dereference on a reference to an object which is *implemented* as a hash itself, the only problem one has to circumvent is how to access this *actual* hash (as opposed to the *virtual* hash exhibited by the overloaded dereference operator). Here is one possible fetching routine:

```

sub access_hash {
 my ($self, $key) = (shift, shift);
 my $class = ref $self;
 bless $self, 'overload::dummy'; # Disable overloading of %{}
 my $out = $self->{$key};
 bless $self, $class; # Restore overloading
 $out;
}

```

To remove creation of the tied hash on each access, one may add an extra level of indirection which allows a non-circular structure of references:

```
package two_refs1;
use overload '%{}' => sub { ${shift()}->[1] },
 '@{}' => sub { ${shift()}->[0] };

sub new {
 my $p = shift;
 my $a = [@_];
 my %h;
 tie %h, $p, $a;
 bless \ [$a, \%h], $p;
}

sub gethash {
 my %h;
 my $self = shift;
 tie %h, ref $self, $self;
 \%h;
}

sub TIEHASH { my $p = shift; bless \ shift, $p }
my %fields;
my $i = 0;
$fields{$_} = $i++ foreach qw{zero one two three};
sub STORE {
 my $a = ${shift()};
 my $key = $fields{shift()};
 defined $key or die "Out of band access";
 $a->[$key] = shift;
}
sub FETCH {
 my $a = ${shift()};
 my $key = $fields{shift()};
 defined $key or die "Out of band access";
 $a->[$key];
}
```

Now if \$baz is overloaded like this, then \$baz is a reference to a reference to the intermediate array, which keeps a reference to an actual array, and the access hash. The tie()ing object for the access hash is a reference to a reference to the actual array, so

- There are no loops of references.
- Both "objects" which are blessed into the class two\_refs1 are references to a reference to an array, thus references to a *scalar*. Thus the accessor expression `$$foo->[$ind]` involves no overloaded operations.

### Symbolic calculator

Put this in *symbolic.pm* in your Perl library directory:

```
package symbolic; # Primitive symbolic calculator
use overload nomethod => \&wrap;

sub new { shift; bless ['n', @_] }
sub wrap {
 my ($obj, $other, $inv, $meth) = @_;
 ($obj, $other) = ($other, $obj) if $inv;
 bless [$meth, $obj, $other];
}
```

```
}
```

This module is very unusual as overloaded modules go: it does not provide any usual overloaded operators, instead it provides the *Last Resort* operator `nomethod`. In this example the corresponding subroutine returns an object which encapsulates operations done over the objects: `new symbolic 3` contains `['n', 3]`, `2 + new symbolic 3` contains `['+', 2, ['n', 3]]`.

Here is an example of the script which "calculates" the side of circumscribed octagon using the above package:

```
require symbolic;
my $iter = 1; # 2**($iter+2) = 8
my $side = new symbolic 1;
my $cnt = $iter;

while ($cnt-- > 0) {
 $side = (sqrt(1 + $side**2) - 1)/$side;
}
print "OK\n";
```

The value of `$side` is

```
['/', ['-', ['sqrt', ['+', 1, ['**', ['n', 1], 2]],
 undef], 1], ['n', 1]]
```

Note that while we obtained this value using a nice little script, there is no simple way to *use* this value. In fact this value may be inspected in debugger (see [perldebug](#)), but only if `bareStringify` Option is set, and not via `p` command.

If one attempts to print this value, then the overloaded operator `"` will be called, which will call `nomethod` operator. The result of this operator will be stringified again, but this result is again of type `symbolic`, which will lead to an infinite loop.

Add a pretty-printer method to the module *symbolic.pm*:

```
sub pretty {
 my ($meth, $a, $b) = @{+shift};
 $a = 'u' unless defined $a;
 $b = 'u' unless defined $b;
 $a = $a->pretty if ref $a;
 $b = $b->pretty if ref $b;
 "[$meth $a $b]";
}
```

Now one can finish the script by

```
print "side = ", $side->pretty, "\n";
```

The method `pretty` is doing object-to-string conversion, so it is natural to overload the operator `"` using this method. However, inside such a method it is not necessary to pretty-print the *components* `$a` and `$b` of an object. In the above subroutine `"[$meth $a $b]"` is a catenation of some strings and components `$a` and `$b`. If these components use overloading, the catenation operator will look for an overloaded operator `.`; if not present, it will look for an overloaded operator `"`. Thus it is enough to use

```
use overload nomethod => \&wrap, '""' => \&str;
sub str {
 my ($meth, $a, $b) = @{+shift};
 $a = 'u' unless defined $a;
 $b = 'u' unless defined $b;
 "[$meth $a $b]";
}
```

Now one can change the last line of the script to

```
print "side = $side\n";
```

which outputs

```
side = [/ [- [sqrt [+ 1 [** [n 1 u] 2]] u] 1] [n 1 u]]
```

and one can inspect the value in debugger using all the possible methods.

Something is still amiss: consider the loop variable `$cnt` of the script. It was a number, not an object. We cannot make this value of type `symbolic`, since then the loop will not terminate.

Indeed, to terminate the cycle, the `$cnt` should become false. However, the operator `bool` for checking falsity is overloaded (this time via overloaded `"`), and returns a long string, thus any object of type `symbolic` is true. To overcome this, we need a way to compare an object to 0. In fact, it is easier to write a numeric conversion routine.

Here is the text of *symbolic.pm* with such a routine added (and slightly modified `str()`):

```
package symbolic; # Primitive symbolic calculator
use overload
 nomethod => \&wrap, '""' => \&str, '0+' => \#

sub new { shift; bless ['n', @_] }
sub wrap {
 my ($obj, $other, $inv, $meth) = @_;
 ($obj, $other) = ($other, $obj) if $inv;
 bless [$meth, $obj, $other];
}
sub str {
 my ($meth, $a, $b) = @{+shift};
 $a = 'u' unless defined $a;
 if (defined $b) {
 "$meth $a $b";
 } else {
 "$meth $a";
 }
}
my %subr = (n => sub {$_[0]},
 sqrt => sub {sqrt $_[0]},
 '-' => sub {shift() - shift()},
 '+' => sub {shift() + shift()},
 '/' => sub {shift() / shift()},
 '*' => sub {shift() * shift()},
 '**' => sub {shift() ** shift()},
);
sub num {
 my ($meth, $a, $b) = @{+shift};
 my $subr = $subr{$meth}
 or die "Do not know how to ($meth) in symbolic";
 $a = $a->num if ref $a eq __PACKAGE__;
 $b = $b->num if ref $b eq __PACKAGE__;
 $subr->($a,$b);
}
```

All the work of numeric conversion is done in `%subr` and `num()`. Of course, `%subr` is not complete, it contains only operators used in the example below. Here is the extra-credit question: why do we need an explicit recursion in `num()`? (Answer is at the end of this section.)

Use this module like this:

```
require symbolic;
my $iter = new symbolic 2; # 16-gon
my $side = new symbolic 1;
my $cnt = $iter;

while ($cnt) {
 $cnt = $cnt - 1; # Mutator '--' not implemented
 $side = (sqrt(1 + $side**2) - 1)/$side;
}
printf "%s=%f\n", $side, $side;
printf "pi=%f\n", $side*(2**($iter+2));
```

It prints (without so many line breaks)

```
[/ [- [sqrt [+ 1 [** [/ [- [sqrt [+ 1 [** [n 1] 2]]] 1]
 [n 1]] 2]]] 1]
 [/ [- [sqrt [+ 1 [** [n 1] 2]]] 1] [n 1]]]=0.198912
pi=3.182598
```

The above module is very primitive. It does not implement mutator methods (++ , -= and so on), does not do deep copying (not required without mutators!), and implements only those arithmetic operations which are used in the example.

To implement most arithmetic operations is easy; one should just use the tables of operations, and change the code which fills %subr to

```
my %subr = ('n' => sub {$_[0]});
foreach my $op (split " ", $overload::ops{with_assign}) {
 $subr{$op} = $subr{"$op="} = eval "sub {shift() $op shift()}";
}
my @bins = qw(binary 3way_comparison num_comparison str_comparison);
foreach my $op (split " ", "@overload::ops{ @bins }") {
 $subr{$op} = eval "sub {shift() $op shift()}";
}
foreach my $op (split " ", "@overload::ops{qw(unary func)}") {
 print "defining '$op'\n";
 $subr{$op} = eval "sub {$op shift()}";
}
```

Due to [Calling Conventions for Mutators](#), we do not need anything special to make += and friends work, except filling += entry of %subr, and defining a copy constructor (needed since Perl has no way to know that the implementation of '+=' does not mutate the argument, compare [Copy Constructor](#)).

To implement a copy constructor, add < '=' = \&cpy to use overload line, and code (this code assumes that mutators change things one level deep only, so recursive copying is not needed):

```
sub cpy {
 my $self = shift;
 bless [@$self], ref $self;
}
```

To make ++ and -- work, we need to implement actual mutators, either directly, or in nomethod. We continue to do things inside nomethod, thus add

```
if ($meth eq '++' or $meth eq '--') {
 @$obj = ($meth, (bless [@$obj]), 1); # Avoid circular reference
 return $obj;
}
```

after the first line of `wrap()`. This is not a most effective implementation, one may consider

```
sub inc { $_[0] = bless ['++', shift, 1]; }
```

instead.

As a final remark, note that one can fill `%subr` by

```
my %subr = ('n' => sub { $_[0] });
foreach my $op (split " ", $overload::ops{with_assign}) {
 $subr{$op} = $subr{"$op="} = eval "sub {shift() $op shift()}";
}
my @bins = qw(binary 3way_comparison num_comparison str_comparison);
foreach my $op (split " ", "@overload::ops{ @bins }") {
 $subr{$op} = eval "sub {shift() $op shift()}";
}
foreach my $op (split " ", "@overload::ops{qw(unary func)}") {
 $subr{$op} = eval "sub { $op shift()}";
}
$subr{'++'} = $subr{'+'};
$subr{'--'} = $subr{'-'};
```

This finishes implementation of a primitive symbolic calculator in 50 lines of Perl code. Since the numeric values of subexpressions are not cached, the calculator is very slow.

Here is the answer for the exercise: In the case of `str()`, we need no explicit recursion since the overloaded `.-` operator will fall back to an existing overloaded operator `"."`. Overloaded arithmetic operators *do not* fall back to numeric conversion if `fallback` is not explicitly requested. Thus without an explicit recursion `num()` would convert `['+', $a, $b]` to `$a + $b`, which would just rebuild the argument of `num()`.

If you wonder why defaults for conversion are different for `str()` and `num()`, note how easy it was to write the symbolic calculator. This simplicity is due to an appropriate choice of defaults. One extra note: due to the explicit recursion `num()` is more fragile than `sym()`: we need to explicitly check for the type of `$a` and `$b`. If components `$a` and `$b` happen to be of some related type, this may lead to problems.

### Really symbolic calculator

One may wonder why we call the above calculator symbolic. The reason is that the actual calculation of the value of expression is postponed until the value is *used*.

To see it in action, add a method

```
sub STORE {
 my $obj = shift;
 $$obj = 1;
 @$obj->[0,1] = ('=', shift);
}
```

to the package `symbolic`. After this change one can do

```
my $a = new symbolic 3;
my $b = new symbolic 4;
my $c = sqrt($a**2 + $b**2);
```

and the numeric value of `$c` becomes 5. However, after calling

```
$a->STORE(12); $b->STORE(5);
```

the numeric value of `$c` becomes 13. There is no doubt now that the module `symbolic` provides a *symbolic* calculator indeed.

To hide the rough edges under the hood, provide a `tie()`d interface to the package `symbolic` (compare with [Metaphor clash](#)). Add methods

```
sub TIESCALAR { my $pack = shift; $pack->new(@_) }
sub FETCH { shift }
sub nop { } # Around a bug
```

(the bug is described in *"BUGS"*). One can use this new interface as

```
tie $a, 'symbolic', 3;
tie $b, 'symbolic', 4;
$a->nop; $b->nop; # Around a bug

my $c = sqrt($a**2 + $b**2);
```

Now numeric value of `$c` is 5. After `$a = 12; $b = 5` the numeric value of `$c` becomes 13. To insulate the user of the module add a method

```
sub vars { my $p = shift; tie($_, $p), $_->nop foreach @_; }
```

Now

```
my ($a, $b);
symbolic->vars($a, $b);
my $c = sqrt($a**2 + $b**2);

$a = 3; $b = 4;
printf "c5 %s=%f\n", $c, $c;

$a = 12; $b = 5;
printf "c13 %s=%f\n", $c, $c;
```

shows that the numeric value of `$c` follows changes to the values of `$a` and `$b`.

## AUTHOR

Ilya Zakharevich <[ilya@math.mps.ohio-state.edu](mailto:ilya@math.mps.ohio-state.edu)>.

## DIAGNOSTICS

When Perl is run with the **-Do** switch or its equivalent, overloading induces diagnostic messages.

Using the `m` command of Perl debugger (see [perldebug](#)) one can deduce which operations are overloaded (and which ancestor triggers this overloading). Say, if `eq` is overloaded, then the method `(eq)` is shown by debugger. The method `()` corresponds to the `fallback` key (in fact a presence of this method shows that this package has overloading enabled, and it is what is used by the `Overloaded` function of module `overload`).

The module might issue the following warnings:

### Odd number of arguments for overload::constant

(W) The call to `overload::constant` contained an odd number of arguments. The arguments should come in pairs.

### '%s' is not an overloadable type

(W) You tried to overload a constant type the `overload` package is unaware of.

### '%s' is not a code reference

(W) The second (fourth, sixth, ...) argument of `overload::constant` needs to be a code reference. Either an anonymous subroutine, or a reference to a subroutine.

## BUGS

Because it is used for overloading, the per-package hash `%OVERLOAD` now has a special meaning in Perl. The symbol table is filled with names looking like line-noise.

For the purpose of inheritance every overloaded package behaves as if `fallback` is present (possibly undefined). This may create interesting effects if some package is not overloaded, but inherits from two overloaded packages.

Relation between overloading and `tie()`ing is broken. Overloading is triggered or not basing on the *previous* class of `tie()`d value.

This happens because the presence of overloading is checked too early, before any `tie()`d access is attempted. If the `FETCH()`ed class of the `tie()`d value does not change, a simple workaround is to access the value immediately after `tie()`ing, so that after this call the *previous* class coincides with the current one.

**Needed:** a way to fix this without a speed penalty.

Barewords are not covered by overloaded string constants.

This document is confusing. There are grammos and misleading language used in places. It would seem a total rewrite is needed.



**NAME**

PerlIO – On demand loader for PerlIO layers and root of PerlIO::\* name space

**SYNOPSIS**

```
open($fh,"<:crlf", "my.txt"); # portably open a text file for reading
open($fh,"<", "his.jpg"); # portably open a binary file for reading
binmode($fh);

Shell:
 PERLIO=perlio perl
```

**DESCRIPTION**

When an undefined layer 'foo' is encountered in an open or binmode layer specification then C code performs the equivalent of:

```
use PerlIO 'foo';
```

The perl code in PerlIO.pm then attempts to locate a layer by doing

```
require PerlIO::foo;
```

Otherwise the PerlIO package is a place holder for additional PerlIO related functions.

The following layers are currently defined:

**unix** Low level layer which calls read, write and lseek etc.

**stdio**

Layer which calls fread, fwrite and fseek/ftell etc. Note that as this is "real" stdio it will ignore any layers beneath it and got straight to the operating system via the C library as usual.

**perlio**

This is a re-implementation of "stdio-like" buffering written as a PerlIO "layer". As such it will call whatever layer is below it for its operations.

**crlf** A layer which does CRLF to "\n" translation distinguishing "text" and "binary" files in the manner of MS-DOS and similar operating systems. (It currently does *not* mimic MS-DOS as far as treating of Control-Z as being an end-of-file marker.)

**utf8** Declares that the stream accepts perl's internal encoding of characters. (Which really is UTF-8 on ASCII machines, but is UTF-EBCDIC on EBCDIC machines.) This allows any character perl can represent to be read from or written to the stream. The UTF-X encoding is chosen to render simple text parts (i.e. non-accented letters, digits and common punctuation) human readable in the encoded file.

Here is how to write your native data out using UTF-8 (or UTF-EBCDIC) and then read it back in.

```
open(F, ">:utf8", "data.utf");
print F $out;
close(F);

open(F, "<:utf8", "data.utf");
$in = <F>;
close(F);
```

**bytes**

This is the inverse of :utf8 layer. It turns off the flag on the layer below so that data read from it is considered to be "octets" i.e. characters in range 0..255 only. Likewise on output perl will warn if a "wide" character is written to a such a stream.

**raw** The `:raw` layer is *defined* as being identical to calling `binmode($fh)` – the stream is made suitable for passing binary data i.e. each byte is passed as-is. The stream will still be buffered. Unlike earlier versions of perl `:raw` is *not* just the inverse of `:crlf` – other layers which would affect the binary nature of the stream are also removed or disabled.

The implementation of `:raw` is as a pseudo-layer which when "pushed" pops itself and then any layers which do not declare themselves as suitable for binary data. (Undoing `:utf8` and `:crlf` are implemented by clearing flags rather than popping layers but that is an implementation detail.)

As a consequence of the fact that `:raw` normally pops layers it usually only makes sense to have it as the only or first element in a layer specification. When used as the first element it provides a known base on which to build e.g.

```
open($fh, ":raw:utf8", ...)
```

will construct a "binary" stream, but then enable UTF-8 translation.

**pop** A pseudo layer that removes the top-most layer. Gives perl code a way to manipulate the layer stack. Should be considered as experimental. Note that `:pop` only works on real layers and will not undo the effects of pseudo layers like `:utf8`. An example of a possible use might be:

```
open($fh, ...)
...
binmode($fh, ":encoding(...)"); # next chunk is encoded
...
binmode($fh, ":pop"); # back to un-encoded
```

A more elegant (and safer) interface is needed.

### Alternatives to raw

To get a binary stream an alternate method is to use:

```
open($fh, "whatever")
binmode($fh);
```

this has advantage of being backward compatible with how such things have had to be coded on some platforms for years.

To get an un-buffered stream specify an unbuffered layer (e.g. `:unix`) in the open call:

```
open($fh, "<:unix", $path)
```

### Defaults and how to override them

If the platform is MS-DOS like and normally does CRLF to "\n" translation for text files then the default layers are :

```
unix crlf
```

(The low level "unix" layer may be replaced by a platform specific low level layer.)

Otherwise if Configure found out how to do "fast" IO using system's stdio, then the default layers are :

```
unix stdio
```

Otherwise the default layers are

```
unix perlIO
```

These defaults may change once perlIO has been better tested and tuned.

The default can be overridden by setting the environment variable `PERLIO` to a space separated list of layers (unix or platform low level layer is always pushed first).

This can be used to see the effect of/bugs in the various layers e.g.

```
cd .../perl/t
PERLIO=stdio ./perl harness
PERLIO=perlio ./perl harness
```

**AUTHOR**

Nick Ing-Simmons <nick@ing-simmons.net>

**SEE ALSO**

*[binmode in perlfunc](#), [open in perlfunc](#), [perlunicode](#), [Encode](#)*

**NAME**

PerlIO::encoding – encoding layer

**SYNOPSIS**

```
open($f, "<:encoding(foo)", "infoo");
open($f, ">:encoding(bar)", "outbar");

use Encode qw(:fallbacks);
$PerlIO::encoding::check = FB_PERLQQ;
```

**DESCRIPTION**

Open a filehandle with a transparent encoding filter.

On input, convert the bytes expected to be in the specified character set and encoding to Perl string data (Unicode and Perl's internal Unicode encoding, UTF-8). On output, convert Perl string data into the specified character set and encoding.

When the layer is pushed the current value of `$PerlIO::encoding::check` is saved and used as the check argument when calling the Encodings encode and decode.

**SEE ALSO**

[open](#), [Encode](#), [binmode](#), [perluniintro](#)

**NAME**

PerlIO::scalar – support module for in-memory IO.

**SYNOPSIS**

```
open($fh, "<", \ $scalar);
open($fh, ">", \ $scalar);

or

open($fh, "<:scalar", \ $scalar);
open($fh, ">:scalar", \ $scalar);
```

**DESCRIPTION**

PerlIO::scalar only exists to use XSLoader to load C code that provides support for treating a scalar as an "in memory" file.

All normal file operations can be performed on the handle. The scalar is considered a stream of bytes. Currently `fileno($fh)` returns `undef`.

**NAME**

PerlIO::via – Helper class for PerlIO layers implemented in perl

**SYNOPSIS**

```
use PerlIO::via::Layer;
open($fh, "<:via(Layer)", ...);

use Some::Other::Package;
open($fh, ">:via(Some::Other::Package)", ...);
```

**DESCRIPTION**

The PerlIO::via module allows you to develop PerlIO layers in Perl, without having to go into the nitty gritty of programming C with XS as the interface to Perl.

One example module, [PerlIO::via::QuotedPrint](#), is included with Perl 5.8.0, and more example modules are available from CPAN, such as [PerlIO::via::StripHTML](#) and [PerlIO::via::Base64](#). The PerlIO::via::StripHTML module for instance, allows you to say:

```
use PerlIO::via::StripHTML;
open(my $fh, "<:via(StripHTML)", "index.html");
my @line = <$fh>;
```

to obtain the text of an HTML-file in an array with all the HTML-tags automagically removed.

Please note that if the layer is created in the PerlIO::via:: namespace, it does **not** have to be fully qualified. The PerlIO::via module will prefix the PerlIO::via:: namespace if the specified modulename does not exist as a fully qualified module name.

**EXPECTED METHODS**

To create a Perl module that implements a PerlIO layer in Perl (as opposed to in C using XS as the interface to Perl), you need to supply some of the following subroutines. It is recommended to create these Perl modules in the PerlIO::via:: namespace, so that they can easily be located on CPAN and use the default namespace feature of the PerlIO::via module itself.

Please note that this is an area of recent development in Perl and that the interface described here is therefore still subject to change (and hopefully will have better documentation and more examples).

In the method descriptions below *\$fh* will be a reference to a glob which can be treated as a perl file handle. It refers to the layer below. *\$fh* is not passed if the layer is at the bottom of the stack, for this reason and to maintain some level of "compatibility" with TIEHANDLE classes it is passed last.

**\$class-PUSHED([\$mode,\$fh])**

Should return an object or the class, or -1 on failure. (Compare TIEHANDLE.) The arguments are an optional mode string ("r", "w", "w+", ...) and a filehandle for the PerlIO layer below. Mandatory.

When layer is pushed as part of an open call, PUSHED will be called *before* the actual open occurs whether than be via OPEN, SYSOPEN, FDOPEM or by letting lower layer do the open.

**\$obj-POPPED([\$fh])**

Optional – layer is about to be removed.

**\$obj-OPEN(\$path,\$mode,\$fh)**

Optional – if not present lower layer does open. If present called for normal opens after layer is pushed. This function is subject to change as there is no easy way to get lower layer to do open and then regain control.

**\$obj-BINMODE([,\$fh])**

Optional – if not available layer is popped on binmode(\$fh) or when :raw is pushed. If present it should return 0 on success -1 on error and undef to pop the layer.

**\$obj-FDOPEN(\$fd[, \$fh])**

Optional – if not present lower layer does open. If present called for opens which pass a numeric file descriptor after layer is pushed. This function is subject to change as there is no easy way to get lower layer to do open and then regain control.

**\$obj-SYSOPEN(\$path, \$imode, \$perm[, \$fh])**

Optional – if not present lower layer does open. If present called for sysopen style opens which pass a numeric mode and permissions after layer is pushed. This function is subject to change as there is no easy way to get lower layer to do open and then regain control.

**\$obj-FILENO(\$fh)**

Returns a numeric value for Unix-like file descriptor. Return -1 if there isn't one. Optional. Default is `fileno($fh)`.

**\$obj-READ(\$buffer, \$len, \$fh)**

Returns the number of octets placed in `$buffer` (must be less than or equal to `$len`). Optional. Default is to use FILL instead.

**\$obj-WRITE(\$buffer, \$fh)**

Returns the number of octets from buffer that have been successfully written.

**\$obj-FILL(\$fh)**

Should return a string to be placed in the buffer. Optional. If not provided must provide READ or reject handles open for reading in PUSHED.

**\$obj-CLOSE(\$fh)**

Should return 0 on success, -1 on error. Optional.

**\$obj-SEEK(\$posn, \$whence, \$fh)**

Should return 0 on success, -1 on error. Optional. Default is to fail, but that is likely to be changed in future.

**\$obj-TELL(\$fh)**

Returns file position. Optional. Default to be determined.

**\$obj-UNREAD(\$buffer, \$fh)**

Returns the number of octets from buffer that have been successfully saved to be returned on future FILL/READ calls. Optional. Default is to push data into a temporary layer above this one.

**\$obj-FLUSH(\$fh)**

Flush any buffered write data. May possibly be called on readable handles too. Should return 0 on success, -1 on error.

**\$obj-SETLINEBUF(\$fh)**

Optional. No return.

**\$obj-CLEARERR(\$fh)**

Optional. No return.

**\$obj-ERROR(\$fh)**

Optional. Returns error state. Default is no error until a mechanism to signal error (die?) is worked out.

**\$obj-EOF(\$fh)**

Optional. Returns end-of-file state. Default is function of return value of FILL or READ.

## EXAMPLES

Check the `PerlIO::via::` namespace on CPAN for examples of PerlIO layers implemented in Perl. To give you an idea how simple the implementation of a PerlIO layer can look, as simple example is included here.

### Example – a Hexadecimal Handle

Given the following module, `PerlIO::via::Hex` :

```
package PerlIO::via::Hex;

sub PUSHED
{
 my ($class,$mode,$fh) = @_;
 # When writing we buffer the data
 my $buf = '';
 return bless \$buf,$class;
}

sub FILL
{
 my ($obj,$fh) = @_;
 my $line = <$fh>;
 return (defined $line) ? pack("H*", $line) : undef;
}

sub WRITE
{
 my ($obj,$buf,$fh) = @_;
 $$obj .= unpack("H*", $buf);
 return length($buf);
}

sub FLUSH
{
 my ($obj,$fh) = @_;
 print $fh $$obj or return -1;
 $$obj = '';
 return 0;
}

1;
```

the following code opens up an output handle that will convert any output to hexadecimal dump of the output bytes: for example "A" will be converted to "41" (on ASCII-based machines, on EBCDIC platforms the "A" will become "c1")

```
use PerlIO::via::Hex;
open(my $fh, ">:via(Hex)", "foo.hex");
```

and the following code will read the hexdump in and convert it on the fly back into bytes:

```
open(my $fh, "<:via(Hex)", "foo.hex");
```



**NAME**

Pod::Checker, podchecker() – check pod documents for syntax errors

**SYNOPSIS**

```
use Pod::Checker;

$syntax_okay = podchecker($filepath, $outputpath, %options);

my $checker = new Pod::Checker %options;
$checker->parse_from_file($filepath, *STDERR);
```

**OPTIONS/ARGUMENTS**

`$filepath` is the input POD to read and `$outputpath` is where to write POD syntax error messages. Either argument may be a scalar indicating a file-path, or else a reference to an open filehandle. If unspecified, the input-file it defaults to `\*STDIN`, and the output-file defaults to `\*STDERR`.

**podchecker()**

This function can take a hash of options:

**-warnings => *val***

Turn warnings on/off. *val* is usually 1 for on, but higher values trigger additional warnings. See ["Warnings"](#).

**DESCRIPTION**

**podchecker** will perform syntax checking of Perl5 POD format documentation.

*NOTE THAT THIS MODULE IS CURRENTLY IN THE BETA STAGE!*

It is hoped that curious/ambitious user will help flesh out and add the additional features they wish to see in **Pod::Checker** and **podchecker** and verify that the checks are consistent with [perlpod](#).

The following checks are currently preformed:

- Unknown ‘=xxxx’ commands, unknown ‘X<...>’ interior-sequences, and unterminated interior sequences.
- Check for proper balancing of =begin and =end. The contents of such a block are generally ignored, i.e. no syntax checks are performed.
- Check for proper nesting and balancing of =over, =item and =back.
- Check for same nested interior-sequences (e.g. L<...L<...>...>).
- Check for malformed or nonexistent entities E<...>.
- Check for correct syntax of hyperlinks L<...>. See [perlpod](#) for details.
- Check for unresolved document-internal links. This check may also reveal misspelled links that seem to be internal links but should be links to something else.

**DIAGNOSTICS****Errors**

- empty =headn

A heading (=head1 or =head2) without any text? That ain’t no heading!

- =over on line *N* without closing =back

The =over command does not have a corresponding =back before the next heading (=head1 or =head2) or the end of the file.

- `=item` without previous `=over`
- `=back` without previous `=over`
  - An `=item` or `=back` command has been found outside a `=over/=back` block.
- No argument for `=begin`
  - A `=begin` command was found that is not followed by the formatter specification.
- `=end` without `=begin`
  - A standalone `=end` command was found.
- Nested `=begin`'s
  - There were at least two consecutive `=begin` commands without the corresponding `=end`. Only one `=begin` may be active at a time.
- `=for` without formatter specification
  - There is no specification of the formatter after the `=for` command.
- unresolved internal link *NAME*
  - The given link to *NAME* does not have a matching node in the current POD. This also happend when a single word node name is not enclosed in " ".
- Unknown command "*CMD*"
  - An invalid POD command has been found. Valid are `=head1`, `=head2`, `=head3`, `=head4`, `=over`, `=item`, `=back`, `=begin`, `=end`, `=for`, `=pod`, `=cut`
- Unknown interior-sequence "*SEQ*"
  - An invalid markup command has been encountered. Valid are: `B<>`, `C<>`, `E<>`, `F<>`, `I<>`, `L<>`, `S<>`, `X<>`, `Z<>`
- nested commands *CMD*<...*CMD*<...>...>
  - Two nested identical markup commands have been found. Generally this does not make sense.
- garbled entity *STRING*
  - The *STRING* found cannot be interpreted as a character entity.
- Entity number out of range
  - An entity specified by number (dec, hex, oct) is out of range (1–255).
- malformed link `L<>`
  - The link found cannot be parsed because it does not conform to the syntax described in [perlpod](#).
- nonempty `Z<>`
  - The `Z<>` sequence is supposed to be empty.
- empty `X<>`
  - The index entry specified contains nothing but whitespace.
- Spurious text after `=pod` / `=cut`
  - The commands `=pod` and `=cut` do not take any arguments.
- Spurious character(s) after `=back`
  - The `=back` command does not take any arguments.

## Warnings

These may not necessarily cause trouble, but indicate mediocre style.

- multiple occurrence of link target *name*
  - The POD file has some `=item` and/or `=head` commands that have the same text. Potential hyperlinks to such a text cannot be unique then.

- line containing nothing but whitespace in paragraph
 

There is some whitespace on a seemingly empty line. POD is very sensitive to such things, so this is flagged. **vi** users switch on the **list** option to avoid this problem.

```
=begin _disabled_
```
- file does not start with =head
 

The file starts with a different POD directive than head. This is most probably something you do not want.

```
=end _disabled_
```
- previous =item has no contents
 

There is a list =item right above the flagged line that has no text contents. You probably want to delete empty items.
- preceding non-item paragraph(s)
 

A list introduced by =over starts with a text or verbatim paragraph, but continues with =items. Move the non-item paragraph out of the =over/=back block.
- =item type mismatch (*one* vs. *two*)
 

A list started with e.g. a bulleted =item and continued with a numbered one. This is obviously inconsistent. For most translators the type of the *first* =item determines the type of the list.
- *N* unescaped <> in paragraph
 

Angle brackets not written as <lt> and <gt> can potentially cause errors as they could be misinterpreted as markup commands. This is only printed when the -warnings level is greater than 1.
- Unknown entity
 

A character entity was found that does not belong to the standard ISO set or the POD specials verbar and sol.
- No items in =over
 

The list opened with =over does not contain any items.
- No argument for =item
 

=item without any parameters is deprecated. It should either be followed by \* to indicate an unordered list, by a number (optionally followed by a dot) to indicate an ordered (numbered) list or simple text for a definition list.
- empty section in previous paragraph
 

The previous section (introduced by a =head command) does not contain any text. This usually indicates that something is missing. Note: A =head1 followed immediately by =head2 does not trigger this warning.
- Verbatim paragraph in NAME section
 

The NAME section (=head1 NAME) should consist of a single paragraph with the script/module name, followed by a dash '-' and a very short description of what the thing is good for.

## Hyperlinks

There are some warnings wrt. malformed hyperlinks.

- ignoring leading/trailing whitespace in link
 

There is whitespace at the beginning or the end of the contents of L<...>.
- (section) in '\$page' deprecated
 

There is a section detected in the page name of L<...>, e.g. L<passwd(2)>. POD hyperlinks may point to POD documents only. Please write C<passwd(2)> instead. Some formatters are able to expand this to appropriate code. For links to (builtin) functions, please say L<perlfunc/mkdir>.

without ( ).

- alternative text/node '%s' contains non-escaped | or /

The characters | and / are special in the L<...> context. Although the hyperlink parser does its best to determine which "/" is text and which is a delimiter in case of doubt, one ought to escape these literal characters like this:

```
/ E<sol>
| E<verbar>
```

## RETURN VALUE

**podchecker** returns the number of POD syntax errors found or -1 if there were no POD commands at all found in the file.

## EXAMPLES

*[T.B.D.]*

## INTERFACE

While checking, this module collects document properties, e.g. the nodes for hyperlinks (=headX, =item) and index entries (X<>). POD translators can use this feature to syntax-check and get the nodes in a first pass before actually starting to convert. This is expensive in terms of execution time, but allows for very robust conversions.

`Pod::Checker->new( %options )`

Return a reference to a new Pod::Checker object that inherits from Pod::Parser and is used for calling the required methods later. The following options are recognized:

`-warnings => num`

Print warnings if num is true. The higher the value of num, the more warnings are printed. Currently there are only levels 1 and 2.

`-quiet => num`

If num is true, do not print any errors/warnings. This is useful when Pod::Checker is used to munge POD code into plain text from within POD formatters.

`$checker->poderror( @args )`

`$checker->poderror( {%opts}, @args )`

Internal method for printing errors and warnings. If no options are given, simply prints "@\_". The following options are recognized and used to form the output:

`-msg`

A message to print prior to @args.

`-line`

The line number the error occurred in.

`-file`

The file (name) the error occurred in.

`-severity`

The error level, should be 'WARNING' or 'ERROR'.

`$checker->num_errors()`

Set (if argument specified) and retrieve the number of errors found.

`$checker->name()`

Set (if argument specified) and retrieve the canonical name of POD as found in the =head1 NAME section.

**`$checker->node()`**

Add (if argument specified) and retrieve the nodes (as defined by `=headX` and `=item`) of the current POD. The nodes are returned in the order of their occurrence. They consist of plain text, each piece of whitespace is collapsed to a single blank.

**`$checker->idx()`**

Add (if argument specified) and retrieve the index entries (as defined by `X<>`) of the current POD. They consist of plain text, each piece of whitespace is collapsed to a single blank.

**`$checker->hyperlink()`**

Add (if argument specified) and retrieve the hyperlinks (as defined by `L<>`) of the current POD. They consist of a 2-item array: line number and `Pod::Hyperlink` object.

**AUTHOR**

Brad Appleton <bradapp@enteract.com> (initial version), Marek Rouchal  
<marek@saftsack.fs.uni-bayreuth.de>

Based on code for **`Pod::Text::pod2text()`** written by Tom Christiansen <tchrist@mox.perl.com>

**NAME**

Pod::Find – find POD documents in directory trees

**SYNOPSIS**

```
use Pod::Find qw(pod_find simplify_name);
my %pods = pod_find({ -verbose => 1, -inc => 1 });
foreach(keys %pods) {
 print "found library POD '$pods{$_}' in $_\n";
}

print "podname=",simplify_name('a/b/c/mymodule.pod'),"\n";

$location = pod_where({ -inc => 1 }, "Pod::Find");
```

**DESCRIPTION**

**Pod::Find** provides a set of functions to locate POD files. Note that no function is exported by default to avoid pollution of your namespace, so be sure to specify them in the **use** statement if you need them:

```
use Pod::Find qw(pod_find);
```

**pod\_find( { %opts } , @directories )**

The function **pod\_find** searches for POD documents in a given set of files and/or directories. It returns a hash with the file names as keys and the POD name as value. The POD name is derived from the file name and its position in the directory tree.

E.g. when searching in *\$HOME/perl5lib*, the file *\$HOME/perl5lib/MyModule.pm* would get the POD name *MyModule*, whereas *\$HOME/perl5lib/Myclass/Subclass.pm* would be *Myclass::Subclass*. The name information can be used for POD translators.

Only text files containing at least one valid POD command are found.

A warning is printed if more than one POD file with the same POD name is found, e.g. *CPAN.pm* in different directories. This usually indicates duplicate occurrences of modules in the *@INC* search path.

**OPTIONS** The first argument for **pod\_find** may be a hash reference with options. The rest are either directories that are searched recursively or files. The POD names of files are the plain basenames with any Perl-like extension (.pm, .pl, .pod) stripped.

**-verbose => 1**

Print progress information while scanning.

**-perl => 1**

Apply Perl-specific heuristics to find the correct PODs. This includes stripping Perl-like extensions, omitting subdirectories that are numeric but do *not* match the current Perl interpreter's version id, suppressing *site\_perl* as a module hierarchy name etc.

**-script => 1**

Search for PODs in the current Perl interpreter's installation **scriptdir**. This is taken from the local *Config/Config* module.

**-inc => 1**

Search for PODs in the current Perl interpreter's *@INC* paths. This automatically considers paths specified in the PERL5LIB environment as this is prepended to *@INC* by the Perl interpreter itself.

**simplify\_name( \$str )**

The function **simplify\_name** is equivalent to **basename**, but also strips Perl-like extensions (.pm, .pl, .pod) and extensions like *.bat*, *.cmd* on Win32 and OS/2, or *.com* on VMS, respectively.

**pod\_where( { %opts }, \$pod )**

Returns the location of a pod document given a search directory and a module (e.g. `File::Find`) or script (e.g. `perldoc`) name.

Options:

`-inc => 1`

Search `@INC` for the pod and also the `scriptdir` defined in the [Config/Config](#) module.

`-dirs => [ $dir1, $dir2, ... ]`

Reference to an array of search directories. These are searched in order before looking in `@INC` (if `-inc`). Current directory is used if none are specified.

`-verbose => 1`

List directories as they are searched

Returns the full path of the first occurrence to the file. Package names (eg `'A::B'`) are automatically converted to directory names in the selected directory. (eg on unix `'A::B'` is converted to `'A/B'`). Additionally, `'.pm'`, `'.pl'` and `'.pod'` are appended to the search automatically if required.

A subdirectory ***pod/*** is also checked if it exists in any of the given search directories. This ensures that e.g. [perlfunc/perlfunc](#) is found.

It is assumed that if a module name is supplied, that that name matches the file name. Pods are not opened to check for the `'NAME'` entry.

A check is made to make sure that the file that is found does contain some pod documentation.

**contains\_pod( \$file , \$verbose )**

Returns true if the supplied filename (not POD module) contains some pod information.

**AUTHOR**

Marek Rouchal <marek@saftsack.fs.uni-bayreuth.de>, heavily borrowing code from Nick Ing-Simmons' PodToHtml.

Tim Jenness <t.jenness@jach.hawaii.edu> provided `pod_where` and `contains_pod`.

**SEE ALSO**

[Pod::Parser](#), [Pod::Checker](#), [perldoc](#)

**NAME**

Pod::Functions – Group Perl’s functions a la perlfunc.pod

**SYNOPSIS**

```
use Pod::Functions;

my @misc_ops = @{ $Kinds{ 'Misc' } };
my $misc_dsc = $Type_Description{ 'Misc' };

or

perl /path/to/lib/Pod/Functions.pm
```

This will print a grouped list of Perl’s functions, like the *[Perl Functions by Category in perlfunc](#)* section.

**DESCRIPTION**

It exports the following variables:

**%Kinds**

This holds a hash-of-lists. Each list contains the functions in the catagory the key denotes.

**%Type**

In this hash each key represents a function and the value is the catagory. The catagory can be a comma separated list.

**%Flavor**

In this hash each key represents a function and the value is a short description of that function.

**%Type\_Description**

In this hash each key represents a catagory of functions and the value is a short description of that catagory.

**@Type\_Order**

This list of catagories is used to produce the same order as the *[Perl Functions by Category in perlfunc](#)* section.

**CHANGES**

1.01 20011229 <abe@ztreet.demon.nl>  
fixed some bugs that slipped in after 5.6.1  
added the pod  
finished making it strict safe

1.00 ??  
first numbered version



**NAME**

Pod::Html – module to convert pod files to HTML

**SYNOPSIS**

```
use Pod::Html;
pod2html([options]);
```

**DESCRIPTION**

Converts files from pod format (see [perlpod](#)) to HTML format. It can automatically generate indexes and cross-references, and it keeps a cache of things it knows how to cross-reference.

**ARGUMENTS**

Pod::Html takes the following arguments:

**backlink**

```
--backlink="Back to Top"
```

Adds "Back to Top" links in front of every `head1` heading (except for the first). By default, no backlinks are generated.

**cachedir**

```
--cachedir=name
```

Creates the item and directory caches in the given directory.

**css**

```
--css=stylesheet
```

Specify the URL of a cascading style sheet. Also disables all HTML/CSS `style` attributes that are output by default (to avoid conflicts).

**flush**

```
--flush
```

Flushes the item and directory caches.

**header**

```
--header
--noheader
```

Creates header and footer blocks containing the text of the NAME section. By default, no headers are generated.

**help**

```
--help
```

Displays the usage message.

**htmldir**

```
--htmldir=name
```

Sets the directory in which the resulting HTML file is placed. This is used to generate relative links to other files. Not passing this causes all links to be absolute, since this is the value that tells Pod::Html the root of the documentation tree.

**htmlroot**

```
--htmlroot=name
```

Sets the base URL for the HTML files. When cross-references are made, the HTML root is prepended to the URL.

**index**

`--index`  
`--noindex`

Generate an index at the top of the HTML file. This is the default behaviour.

**infile**

`--infile=name`

Specify the pod file to convert. Input is taken from STDIN if no infile is specified.

**libpods**

`--libpods=name:...:name`

List of page names (eg, "perlfunc") which contain linkable `=items`.

**netscape**

`--netscape`  
`--nonetscape`

**Deprecated**, has no effect. For backwards compatibility only.

**outfile**

`--outfile=name`

Specify the HTML file to create. Output goes to STDOUT if no outfile is specified.

**podpath**

`--podpath=name:...:name`

Specify which subdirectories of the podroot contain pod files whose HTML converted forms can be linked to in cross references.

**podroot**

`--podroot=name`

Specify the base directory for finding library pods.

**quiet**

`--quiet`  
`--noquiet`

Don't display *mostly harmless* warning messages. These messages will be displayed by default. But this is not the same as verbose mode.

**recurse**

`--recurse`  
`--norecurse`

Recurse into subdirectories specified in podpath (default behaviour).

**title**

`--title=title`

Specify the title of the resulting HTML file.

**verbose**

`--verbose`  
`--noverbose`

Display progress messages. By default, they won't be displayed.

**EXAMPLE**

```
pod2html ("pod2html",
 "--podpath=lib:ext:pod:vms",
 "--podroot=/usr/src/perl",
 "--htmlroot=/perl/nmanual",
 "--libpods=perlfunc:perlguts:perlvar:perlrun:perlop",
 "--recurse",
 "--infile=foo.pod",
 "--outfile=/perl/nmanual/foo.html");
```

**ENVIRONMENT**

Uses `$Config{pod2html}` to setup default options.

**AUTHOR**

Tom Christiansen, <tchrist@perl.com>.

**SEE ALSO**

*[perlpod](#)*

**COPYRIGHT**

This program is distributed under the Artistic License.

**NAME**

Pod::InputObjects – objects representing POD input paragraphs, commands, etc.

**SYNOPSIS**

```
use Pod::InputObjects;
```

**REQUIRES**

perl5.004, Carp

**EXPORTS**

Nothing.

**DESCRIPTION**

This module defines some basic input objects used by **Pod::Parser** when reading and parsing POD text from an input source. The following objects are defined:

```
=begin __PRIVATE__
```

**package Pod::InputSource**

An object corresponding to a source of POD input text. It is mostly a wrapper around a filehandle or `IO::Handle`-type object (or anything that implements the `getline()` method) which keeps track of some additional information relevant to the parsing of PODs.

```
=end __PRIVATE__
```

**package Pod::Paragraph**

An object corresponding to a paragraph of POD input text. It may be a plain paragraph, a verbatim paragraph, or a command paragraph (see [perlpod](#)).

**package Pod::InteriorSequence**

An object corresponding to an interior sequence command from the POD input text (see [perlpod](#)).

**package Pod::ParseTree**

An object corresponding to a tree of parsed POD text. Each "node" in a parse-tree (or *ptree*) is either a text-string or a reference to a **Pod::InteriorSequence** object. The nodes appear in the parse-tree in the order in which they were parsed from left-to-right.

Each of these input objects are described in further detail in the sections which follow.

**Pod::InputSource**

This object corresponds to an input source or stream of POD documentation. When parsing PODs, it is necessary to associate and store certain context information with each input source. All of this information is kept together with the stream itself in one of these `Pod::InputSource` objects. Each such object is merely a wrapper around an `IO::Handle` object of some kind (or at least something that implements the `getline()` method). They have the following methods/attributes:

```
=end __PRIVATE__
```

**new()**

```
my $pod_input1 = Pod::InputSource->new(-handle => $filehandle);
my $pod_input2 = new Pod::InputSource(-handle => $filehandle,
 -name => $name);
my $pod_input3 = new Pod::InputSource(-handle => *STDIN);
my $pod_input4 = Pod::InputSource->new(-handle => *STDIN,
 -name => "(STDIN)");
```

This is a class method that constructs a `Pod::InputSource` object and returns a reference to the new input source object. It takes one or more keyword arguments in the form of a hash. The keyword `-handle` is required and designates the corresponding input handle. The keyword `-name` is optional and specifies the

name associated with the input handle (typically a file name).

```
=end __PRIVATE__
```

#### **name()**

```
my $filename = $pod_input->name();
$pod_input->name($new_filename_to_use);
```

This method gets/sets the name of the input source (usually a filename). If no argument is given, it returns a string containing the name of the input source; otherwise it sets the name of the input source to the contents of the given argument.

```
=end __PRIVATE__
```

#### **handle()**

```
my $handle = $pod_input->handle();
```

Returns a reference to the handle object from which input is read (the one used to construct this input source object).

```
=end __PRIVATE__
```

#### **was\_cutting()**

```
print "Yes.\n" if ($pod_input->was_cutting());
```

The value of the cutting state (that the **cutting()** method would have returned) immediately before any input was read from this input stream. After all input from this stream has been read, the cutting state is restored to this value.

```
=end __PRIVATE__
```

### **Pod::Paragraph**

An object representing a paragraph of POD input text. It has the following methods/attributes:

#### **Pod::Paragraph->new()**

```
my $pod_para1 = Pod::Paragraph->new(-text => $text);
my $pod_para2 = Pod::Paragraph->new(-name => $cmd,
 -text => $text);
my $pod_para3 = new Pod::Paragraph(-text => $text);
my $pod_para4 = new Pod::Paragraph(-name => $cmd,
 -text => $text);
my $pod_para5 = Pod::Paragraph->new(-name => $cmd,
 -text => $text,
 -file => $filename,
 -line => $line_number);
```

This is a class method that constructs a **Pod::Paragraph** object and returns a reference to the new paragraph object. It may be given one or two keyword arguments. The **-text** keyword indicates the corresponding text of the POD paragraph. The **-name** keyword indicates the name of the corresponding POD command, such as **head1** or **item** (it should *not* contain the **=** prefix); this is needed only if the POD paragraph corresponds to a command paragraph. The **-file** and **-line** keywords indicate the filename and line number corresponding to the beginning of the paragraph

#### **\$pod\_para->cmd\_name()**

```
my $para_cmd = $pod_para->cmd_name();
```

If this paragraph is a command paragraph, then this method will return the name of the command (*without* any leading **=** prefix).

**\$pod\_para->text()**

```
my $para_text = $pod_para->text();
```

This method will return the corresponding text of the paragraph.

**\$pod\_para->raw\_text()**

```
my $raw_pod_para = $pod_para->raw_text();
```

This method will return the *raw* text of the POD paragraph, exactly as it appeared in the input.

**\$pod\_para->cmd\_prefix()**

```
my $prefix = $pod_para->cmd_prefix();
```

If this paragraph is a command paragraph, then this method will return the prefix used to denote the command (which should be the string "=" or "==").

**\$pod\_para->cmd\_separator()**

```
my $separator = $pod_para->cmd_separator();
```

If this paragraph is a command paragraph, then this method will return the text used to separate the command name from the rest of the paragraph (if any).

**\$pod\_para->parse\_tree()**

```
my $ptree = $pod_parser->parse_text($pod_para->text());
$pod_para->parse_tree($ptree);
$ptree = $pod_para->parse_tree();
```

This method will get/set the corresponding parse-tree of the paragraph's text.

**\$pod\_para->file\_line()**

```
my ($filename, $line_number) = $pod_para->file_line();
my $position = $pod_para->file_line();
```

Returns the current filename and line number for the paragraph object. If called in a list context, it returns a list of two elements: first the filename, then the line number. If called in a scalar context, it returns a string containing the filename, followed by a colon (:), followed by the line number.

**Pod::InteriorSequence**

An object representing a POD interior sequence command. It has the following methods/attributes:

**Pod::InteriorSequence->new()**

```
my $pod_seq1 = Pod::InteriorSequence->new(-name => $cmd
 -ldelim => $delimiter);
my $pod_seq2 = new Pod::InteriorSequence(-name => $cmd,
 -ldelim => $delimiter);
my $pod_seq3 = new Pod::InteriorSequence(-name => $cmd,
 -ldelim => $delimiter,
 -file => $filename,
 -line => $line_number);

my $pod_seq4 = new Pod::InteriorSequence(-name => $cmd, $ptree);
my $pod_seq5 = new Pod::InteriorSequence($cmd, $ptree);
```

This is a class method that constructs a `Pod::InteriorSequence` object and returns a reference to the new interior sequence object. It should be given two keyword arguments. The `-ldelim` keyword indicates the corresponding left-delimiter of the interior sequence (e.g. '<'). The `-name` keyword indicates the name of the corresponding interior sequence command, such as I or B or C. The `-file` and `-line` keywords indicate the filename and line number corresponding to the beginning of the interior sequence. If the `$ptree` argument is given, it must be the last argument, and it must be either string, or else an array-ref suitable for passing to **Pod::ParseTree::new** (or it may be a reference to a `Pod::ParseTree` object).

```
$pod_seq->cmd_name()
```

```
my $seq_cmd = $pod_seq->cmd_name();
```

The name of the interior sequence command.

```
$pod_seq->prepend()
```

```
$pod_seq->prepend($text);
$pod_seq1->prepend($pod_seq2);
```

Prepends the given string or parse-tree or sequence object to the parse-tree of this interior sequence.

```
$pod_seq->append()
```

```
$pod_seq->append($text);
$pod_seq1->append($pod_seq2);
```

Appends the given string or parse-tree or sequence object to the parse-tree of this interior sequence.

```
$pod_seq->nested()
```

```
$outer_seq = $pod_seq->nested || print "not nested";
```

If this interior sequence is nested inside of another interior sequence, then the outer/parent sequence that contains it is returned. Otherwise undef is returned.

```
$pod_seq->raw_text()
```

```
my $seq_raw_text = $pod_seq->raw_text();
```

This method will return the *raw* text of the POD interior sequence, exactly as it appeared in the input.

```
$pod_seq->left_delimiter()
```

```
my $ldelim = $pod_seq->left_delimiter();
```

The leftmost delimiter beginning the argument text to the interior sequence (should be "<").

```
$pod_seq->right_delimiter()
```

The rightmost delimiter beginning the argument text to the interior sequence (should be "").

```
$pod_seq->parse_tree()
```

```
my $ptree = $pod_parser->parse_text($paragraph_text);
$pod_seq->parse_tree($ptree);
$ptree = $pod_seq->parse_tree();
```

This method will get/set the corresponding parse-tree of the interior sequence's text.

```
$pod_seq->file_line()
```

```
my ($filename, $line_number) = $pod_seq->file_line();
my $position = $pod_seq->file_line();
```

Returns the current filename and line number for the interior sequence object. If called in a list context, it returns a list of two elements: first the filename, then the line number. If called in a scalar context, it returns a string containing the filename, followed by a colon (':'), followed by the line number.

```
Pod::InteriorSequence::DESTROY()
```

This method performs any necessary cleanup for the interior-sequence. If you override this method then it is **imperative** that you invoke the parent method from within your own method, otherwise *interior-sequence storage will not be reclaimed upon destruction!*

```
Pod::ParseTree
```

This object corresponds to a tree of parsed POD text. As POD text is scanned from left to right, it is parsed into an ordered list of text-strings and **Pod::InteriorSequence** objects (in order of appearance). A **Pod::ParseTree** object corresponds to this list of strings and sequences. Each interior sequence in the parse-tree may itself contain a parse-tree (since interior sequences may be nested).

**Pod::ParseTree->new()**

```
my $ptree1 = Pod::ParseTree->new;
my $ptree2 = new Pod::ParseTree;
my $ptree4 = Pod::ParseTree->new($array_ref);
my $ptree3 = new Pod::ParseTree($array_ref);
```

This is a class method that constructs a `Pod::Parse_tree` object and returns a reference to the new parse-tree. If a single-argument is given, it must be a reference to an array, and is used to initialize the root (top) of the parse tree.

**\$ptree->top()**

```
my $top_node = $ptree->top();
$ptree->top($top_node);
$ptree->top(@children);
```

This method gets/sets the top node of the parse-tree. If no arguments are given, it returns the topmost node in the tree (the root), which is also a **Pod::ParseTree**. If it is given a single argument that is a reference, then the reference is assumed to a parse-tree and becomes the new top node. Otherwise, if arguments are given, they are treated as the new list of children for the top node.

**\$ptree->children()**

This method gets/sets the children of the top node in the parse-tree. If no arguments are given, it returns the list (array) of children (each of which should be either a string or a **Pod::InteriorSequence**. Otherwise, if arguments are given, they are treated as the new list of children for the top node.

**\$ptree->prepend()**

This method prepends the given text or parse-tree to the current parse-tree. If the first item on the parse-tree is text and the argument is also text, then the text is prepended to the first item (not added as a separate string). Otherwise the argument is added as a new string or parse-tree *before* the current one.

**\$ptree->append()**

This method appends the given text or parse-tree to the current parse-tree. If the last item on the parse-tree is text and the argument is also text, then the text is appended to the last item (not added as a separate string). Otherwise the argument is added as a new string or parse-tree *after* the current one.

**\$ptree->raw\_text()**

```
my $ptree_raw_text = $ptree->raw_text();
```

This method will return the *raw* text of the POD parse-tree exactly as it appeared in the input.

**Pod::ParseTree::DESTROY()**

This method performs any necessary cleanup for the parse-tree. If you override this method then it is **imperative** that you invoke the parent method from within your own method, otherwise *parse-tree storage will not be reclaimed upon destruction!*

**SEE ALSO**

See [Pod::Parser](#), [Pod::Select](#)

**AUTHOR**

Brad Appleton <bradapp@enteract.com>



**NAME**

Pod::LaTeX – Convert Pod data to formatted Latex

**SYNOPSIS**

```
use Pod::LaTeX;
my $parser = Pod::LaTeX->new ();

$parser->parse_from_filehandle;

$parser->parse_from_file ('file.pod', 'file.tex');
```

**DESCRIPTION**

Pod::LaTeX is a module to convert documentation in the Pod format into Latex. The [pod2latex/pod2latex](#) X<pod2latex command uses this module for translation.

Pod::LaTeX is a derived class from [Pod::Select/Pod::Select](#).

**OBJECT METHODS**

The following methods are provided in this module. Methods inherited from Pod::Select are not described in the public interface.

```
=begin __PRIVATE__

initialize

 Initialise the object. This method is subclassed from Pod::Parser. The base class method is
 invoked. This method defines the default behaviour of the object unless overridden by supplying
 arguments to the constructor.

 Internal settings are defaulted as well as the public instance data. Internal hash values are accessed
 directly (rather than through a method) and start with an underscore.

 This method should not be invoked by the user directly.

=end __PRIVATE__
```

**Data Accessors**

The following methods are provided for accessing instance data. These methods should be used for accessing configuration parameters rather than assuming the object is a hash.

Default values can be supplied by using these names as keys to a hash of arguments when using the new() constructor.

**AddPreamble**

Logical to control whether a latex preamble is to be written. If true, a valid latex preamble is written before the pod data is written. This is similar to:

```
\documentclass{article}
\begin{document}
```

but will be more complicated if table of contents and indexing are required. Can be used to set or retrieve the current value.

```
$add = $parser->AddPreamble();
$parser->AddPreamble(1);
```

If used in conjunction with AddPostamble a full latex document will be written that could be immediately processed by latex.

**AddPostamble**

Logical to control whether a standard latex ending is written to the output file after the document has been processed. In its simplest form this is simply:

```
\end{document}
```

but can be more complicated if an index is required. Can be used to set or retrieve the current value.

```
$add = $parser->AddPostamble();
$parser->AddPostamble(1);
```

If used in conjunction with `AddPreamble` a full latex document will be written that could be immediately processed by latex.

### Head1Level

The latex sectioning level that should be used to correspond to a pod `=head1` directive. This can be used, for example, to turn a `=head1` into a latex subsection. This should hold a number corresponding to the required position in an array containing the following elements:

```
[0] chapter
[1] section
[2] subsection
[3] subsubsection
[4] paragraph
[5] subparagraph
```

Can be used to set or retrieve the current value:

```
$parser->Head1Level(2);
$sect = $parser->Head1Level;
```

Setting this number too high can result in sections that may not be reproducible in the expected way. For example, setting this to 4 would imply that `=head3` do not have a corresponding latex section (`=head1` would correspond to a paragraph).

A check is made to ensure that the supplied value is an integer in the range 0 to 5.

Default is for a value of 1 (i.e. a section).

### Label

This is the label that is prefixed to all latex label and index entries to make them unique. In general, pods have similarly titled sections (NAME, DESCRIPTION etc) and a latex label will be multiply defined if more than one pod document is to be included in a single latex file. To overcome this, this label is prefixed to a label whenever a label is required (joined with an underscore) or to an index entry (joined by an exclamation mark which is the normal index separator). For example, `\label{text}` becomes `\label{Label_text}`.

Can be used to set or retrieve the current value:

```
$label = $parser->Label;
$parser->Label($label);
```

This label is only used if `UniqueLabels` is true. Its value is set automatically from the `NAME` field if `ReplaceNAMEwithSection` is true. If this is not the case it must be set manually before starting the parse.

Default value is `undef`.

### LevelNoNum

Control the point at which latex section numbering is turned off. For example, this can be used to make sure that latex sections are numbered but subsections are not.

Can be used to set or retrieve the current value:

```
$lev = $parser->LevelNoNum;
$parser->LevelNoNum(2);
```

The argument must be an integer between 0 and 5 and is the same as the number described in `Head1Level` method description. The number has nothing to do with the pod heading number, only the latex sectioning.

Default is 2. (i.e. latex subsections are written as `subsection*` but sections are numbered).

### MakeIndex

Controls whether latex commands for creating an index are to be inserted into the preamble and postamble

```
$makeindex = $parser->MakeIndex;
$parser->MakeIndex(0);
```

Irrelevant if both `AddPreamble` and `AddPostamble` are false (or equivalently, `UserPreamble` and `UserPostamble` are set).

Default is for an index to be created.

### ReplaceNAMEwithSection

This controls whether the NAME section in the pod is to be translated literally or converted to a slightly modified output where the section name is the pod name rather than "NAME".

If true, the pod segment

```
=head1 NAME
pod::name - purpose
=head1 SYNOPSIS
```

is converted to the latex

```
\section{pod::name\label{pod_name}\index{pod::name}}
Purpose
\subsection*{SYNOPSIS\label{pod_name_SYNOPSIS}%
\index{pod::name!SYNOPSIS}}
```

(dependent on the value of `Head1Level` and `LevelNoNum`). Note that subsequent `head1` directives translate to subsections rather than sections and that the labels and index now include the pod name (dependent on the value of `UniqueLabels`).

The Label is set from the pod name regardless of any current value of `Label`.

```
$mod = $parser->ReplaceNAMEwithSection;
$parser->ReplaceNAMEwithSection(0);
```

Default is to translate the pod literally.

### StartWithNewPage

If true, each pod translation will begin with a latex `\clearpage`.

```
$parser->StartWithNewPage(1);
$newpage = $parser->StartWithNewPage;
```

Default is false.

### TableOfContents

If true, a table of contents will be created. Irrelevant if `AddPreamble` is false or `UserPreamble` is set.

```
$toc = $parser->TableOfContents;
$parser->TableOfContents(1);
```

Default is false.

### UniqueLabels

If true, the translator will attempt to make sure that each `latex` label or index entry will be uniquely identified by prefixing the contents of `Label`. This allows multiple documents to be combined without clashing common labels such as `DESCRIPTION` and `SYNOPSIS`

```
$parser->UniqueLabels(1);
$unq = $parser->UniqueLabels;
```

Default is true.

### UserPreamble

User supplied `latex` preamble. Added before the pod translation data.

If set, the contents will be prepended to the output file before the translated data regardless of the value of `AddPreamble`. `MakeIndex` and `TableOfContents` will also be ignored.

### UserPostamble

User supplied `latex` postamble. Added after the pod translation data.

If set, the contents will be prepended to the output file after the translated data regardless of the value of `AddPostamble`. `MakeIndex` will also be ignored.

### Lists

Contains details of the currently active lists.

The array contains `Pod::List` objects. A new `Pod::List` object is created each time a list is encountered and it is pushed onto this stack. When the list context ends, it is popped from the stack. The array will be empty if no lists are active.

Returns array of list information in list context Returns array ref in scalar context

`=begin __PRIVATE__`

### Subclassed methods

The following methods override methods provided in the `Pod::Select` base class. See `Pod::Parser` and `Pod::Select` for more information on what these methods require.

#### **begin\_pod**

Writes the `latex` preamble if requested.

#### **end\_pod**

Write the closing `latex` code.

#### **command**

Process basic pod commands.

#### **verbatim**

Verbatim text

#### **textblock**

Plain text paragraph.

#### **interior\_sequence**

Interior sequence expansion

### List Methods

Methods used to handle lists.

**begin\_list**

Called when a new list is found (via the `over` directive). Creates a new `Pod::List` object and stores it on the `list` stack.

```
$parser->begin_list($indent, $line_num);
```

**end\_list**

Called when the end of a list is found (the `back` directive). Pops the `Pod::List` object off the stack of lists and writes the `latex` code required to close a list.

```
$parser->end_list($line_num);
```

**add\_item**

Add items to the list. The first time an item is encountered (determined from the state of the current `Pod::List` object) the type of list is determined (ordered, unnumbered or description) and the relevant `latex` code issued.

```
$parser->add_item($paragraph, $line_num);
```

**Methods for headings****head**

Print a heading of the required level.

```
$parser->head($level, $paragraph, $parobj);
```

The first argument is the pod heading level. The second argument is the contents of the heading. The 3rd argument is a `Pod::Paragraph` object so that the line number can be extracted.

```
=end __PRIVATE__
```

```
=begin __PRIVATE__
```

**Internal methods**

Internal routines are described in this section. They do not form part of the public interface. All private methods start with an underscore.

**\_output**

Output text to the output filehandle. This method must be always be called to output parsed text.

```
$parser->_output($text);
```

Does not write anything if a `=begin` is active that should be ignored.

**\_replace\_special\_chars**

Subroutine to replace characters that are special in `latex` with the escaped forms

```
$escaped = $parser->_replace_special_chars($paragraph);
```

Need to call this routine before `interior_sequences` are munged but not if `verbatim`. It must be called before interpolation of interior sequences so that curly brackets and special `latex` characters inserted during interpolation are not themselves escaped. This means that `<` and `&` can not be modified here since the text still contains interior sequences.

Special characters and the `latex` equivalents are:

|    |                |
|----|----------------|
| }  | \}             |
| {  | \{             |
| _  | \_             |
| \$ | \\$            |
| %  | \%             |
| &  | \&             |
| \  | \$\backslash\$ |

```

^ \^{}
~ \~{}

```

### **`_replace_special_chars_late`**

Replace special characters that can not be replaced before interior sequence interpolation. See `_replace_special_chars` for a routine to replace special characters prior to interpolation of interior sequences.

Does the following transformation:

```

< $<$
> $>$
| $|$

```

### **`_create_label`**

Return a string that can be used as an internal reference in a latex document (i.e. accepted by the `\label` command)

```
$label = $parser->_create_label($string)
```

If `UniqueLabels` is true returns a label prefixed by `Label()` This can be suppressed with an optional second argument.

```
$label = $parser->_create_label($string, $suppress);
```

If a second argument is supplied (of any value including undef) the `Label()` is never prefixed. This means that this routine can be called to create a `Label()` without prefixing a previous setting.

### **`_create_index`**

Similar to `_create_label` except an index entry is created. If `UniqueLabels` is true, the index entry is prefixed by the current `Label` and an exclamation mark.

```
$ind = $parser->_create_index($paragraph);
```

An exclamation mark is used by `makeindex` to generate sub-entries in an index.

### **`_clean_latex_commands`**

Removes latex commands from text. The latex command is assumed to be of the form `\command{ text }`. "text" is retained

```
$clean = $parser->_clean_latex_commands($text);
```

### **`_split_delimited`**

Split the supplied string into two parts at approximately the specified word boundary. Special care is made to make sure that it does not split in the middle of some curly brackets.

e.g. "this text is `\textbf{very bold}`" would not be split into "this text is `\textbf{very}`" and " bold".

```
($hunk1, $hunk2) = $self->_split_delimited($para, $length);
```

The length indicates the maximum length of `hunk1`.

```
=end __PRIVATE__
```

## **NOTES**

Compatible with latex2e only. Can not be used with latex v2.09 or earlier.

A subclass of `Pod::Select` so that specific pod sections can be converted to latex by using the `select` method.

Some HTML escapes are missing and many have not been tested.

**SEE ALSO**

*Pod::Parser*, *Pod::Select*, *pod2latex*

**AUTHORS**

Tim Jenness <t.jenness@jach.hawaii.edu>

Bug fixes have been received from: Simon Cozens <simon@cozens.net>, Mark A. Hershberger <mah@everybody.org>, Marcel Grunauer <marcel@codewerk.com> and Hugh S Myers <hsmyers@sdragons.com>.

**COPYRIGHT**

Copyright (C) 2000–2001 Tim Jenness. All Rights Reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

=begin \_\_PRIVATE\_\_

**REVISION**

\$Id: LaTeX.pm,v 1.12 2001/11/20 20:59:26 timj Exp \$

=end \_\_PRIVATE\_\_

**NAME**

Pod::Man – Convert POD data to formatted \*roff input

**SYNOPSIS**

```
use Pod::Man;
my $parser = Pod::Man->new (release => $VERSION, section => 8);

Read POD from STDIN and write to STDOUT.
$parser->parse_from_filehandle;

Read POD from file.pod and write to file.1.
$parser->parse_from_file ('file.pod', 'file.1');
```

**DESCRIPTION**

Pod::Man is a module to convert documentation in the POD format (the preferred language for documenting Perl) into \*roff input using the man macro set. The resulting \*roff code is suitable for display on a terminal using *nroff(1)*, normally via *man(1)*, or printing using *troff(1)*. It is conventionally invoked using the driver script **pod2man**, but it can also be used directly.

As a derived class from Pod::Parser, Pod::Man supports the same methods and interfaces. See *Pod::Parser* for all the details; briefly, one creates a new parser with `< Pod::Man-new()` and then calls either `parse_from_filehandle()` or `parse_from_file()`.

`new()` can take options, in the form of key/value pairs that control the behavior of the parser. See below for details.

If no options are given, Pod::Man uses the name of the input file with any trailing .pod, .pm, or .pl stripped as the man page title, to section 1 unless the file ended in .pm in which case it defaults to section 3, to a centered title of "User Contributed Perl Documentation", to a centered footer of the Perl version it is run with, and to a left-hand footer of the modification date of its input (or the current date if given STDIN for input).

Pod::Man assumes that your \*roff formatters have a fixed-width font named CW. If yours is called something else (like CR), use the `fixed` option to specify it. This generally only matters for troff output for printing. Similarly, you can set the fonts used for bold, italic, and bold italic fixed-width output.

Besides the obvious pod conversions, Pod::Man also takes care of formatting `func()`, `func(3)`, and simple variable references like `$foo` or `@bar` so you don't have to use code escapes for them; complex expressions like `$fred{'stuff'}` will still need to be escaped, though. It also translates dashes that aren't used as hyphens into en dashes, makes long dashes—like this—into proper em dashes, fixes "paired quotes," makes C++ look right, puts a little space between double underbars, makes ALLCAPS a teeny bit smaller in **troff**, and escapes stuff that \*roff treats as special so that you don't have to.

The recognized options to `new()` are as follows. All options take a single argument.

**center**

Sets the centered page header to use instead of "User Contributed Perl Documentation".

**date**

Sets the left-hand footer. By default, the modification date of the input file will be used, or the current date if `stat()` can't find that file (the case if the input is from STDIN), and the date will be formatted as YYYY-MM-DD.

**fixed**

The fixed-width font to use for verbatim text and code. Defaults to CW. Some systems may want CR instead. Only matters for **troff** output.

**fixedbold**

Bold version of the fixed-width font. Defaults to CB. Only matters for **troff** output.



**fixeditalic**

Italic version of the fixed-width font (actually, something of a misnomer, since most fixed-width fonts only have an oblique version, not an italic version). Defaults to CI. Only matters for **troff** output.

**fixedbolditalic**

Bold italic (probably actually oblique) version of the fixed-width font. Pod::Man doesn't assume you have this, and defaults to CB. Some systems (such as Solaris) have this font available as CX. Only matters for **troff** output.

**name**

Set the name of the manual page. Without this option, the manual name is set to the uppercased base name of the file being converted unless the manual section is 3, in which case the path is parsed to see if it is a Perl module path. If it is, a path like `.../lib/Pod/Man.pm` is converted into a name like `Pod::Man`. This option, if given, overrides any automatic determination of the name.

**quotes**

Sets the quote marks used to surround `C< text`. If the value is a single character, it is used as both the left and right quote; if it is two characters, the first character is used as the left quote and the second as the right quoted; and if it is four characters, the first two are used as the left quote and the second two as the right quote.

This may also be set to the special value `none`, in which case no quote marks are added around `C< text` (but the font is still changed for troff output).

**release**

Set the centered footer. By default, this is the version of Perl you run Pod::Man under. Note that some system an macro sets assume that the centered footer will be a modification date and will prepend something like "Last modified: "; if this is the case, you may want to set `release` to the last modified date and `date` to the version number.

**section**

Set the section for the `.TH` macro. The standard section numbering convention is to use 1 for user commands, 2 for system calls, 3 for functions, 4 for devices, 5 for file formats, 6 for games, 7 for miscellaneous information, and 8 for administrator commands. There is a lot of variation here, however; some systems (like Solaris) use 4 for file formats, 5 for miscellaneous information, and 7 for devices. Still others use 1m instead of 8, or some mix of both. About the only section numbers that are reliably consistent are 1, 2, and 3.

By default, section 1 will be used unless the file ends in `.pm` in which case section 3 will be selected.

The standard Pod::Parser method `parse_from_filehandle()` takes up to two arguments, the first being the file handle to read POD from and the second being the file handle to write the formatted output to. The first defaults to `STDIN` if not given, and the second defaults to `STDOUT`. The method `parse_from_file()` is almost identical, except that its two arguments are the input and output disk files instead. See [Pod::Parser](#) for the specific details.

**DIAGNOSTICS**

roff font should be 1 or 2 chars, not "%s"

(F) You specified a \*roff font (using `fixed`, `fixedbold`, etc.) that wasn't either one or two characters. Pod::Man doesn't support \*roff fonts longer than two characters, although some \*roff extensions do (the canonical versions of **nroff** and **troff** don't either).

Invalid link %s

(W) The POD source contained a `L<>` formatting code that Pod::Man was unable to parse. You should never see this error message; it probably indicates a bug in Pod::Man.

**Invalid quote specification "%s"**

(F) The quote specification given (the quotes option to the constructor) was invalid. A quote specification must be one, two, or four characters long.

**%s:%d: Unknown command paragraph "%s".**

(W) The POD source contained a non-standard command paragraph (something of the form `=command args`) that Pod::Man didn't know about. It was ignored.

**%s:%d: Unknown escape E<%s>**

(W) The POD source contained an E<> escape that Pod::Man didn't know about. E<%s> was printed verbatim in the output.

**%s:%d: Unknown formatting code %s**

(W) The POD source contained a non-standard formatting code (something of the form X<>) that Pod::Man didn't know about. It was ignored.

**%s:%d: Unmatched =back**

(W) Pod::Man encountered a `=back` command that didn't correspond to an `=over` command.

**BUGS**

Eight-bit input data isn't handled at all well at present. The correct approach would be to map E<> escapes to the appropriate UTF-8 characters and then do a translation pass on the output according to the user-specified output character set. Unfortunately, we can't send eight-bit data directly to the output unless the user says this is okay, since some vendor \*roff implementations can't handle eight-bit data. If the \*roff implementation can, however, that's far superior to the current hacked characters that only work under troff.

There is currently no way to turn off the guesswork that tries to format unmarked text appropriately, and sometimes it isn't wanted (particularly when using POD to document something other than Perl).

The NAME section should be recognized specially and index entries emitted for everything in that section. This would have to be deferred until the next section, since extraneous things in NAME tends to confuse various man page processors.

Pod::Man doesn't handle font names longer than two characters. Neither do most **troff** implementations, but GNU troff does as an extension. It would be nice to support as an option for those who want to use it.

The preamble added to each output file is rather verbose, and most of it is only necessary in the presence of E<> escapes for non-ASCII characters. It would ideally be nice if all of those definitions were only output if needed, perhaps on the fly as the characters are used.

Pod::Man is excessively slow.

**CAVEATS**

The handling of hyphens and em dashes is somewhat fragile, and one may get the wrong one under some circumstances. This should only matter for **troff** output.

When and whether to use small caps is somewhat tricky, and Pod::Man doesn't necessarily get it right.

**SEE ALSO**

*Pod::Parser*, *perlpod(1)*, *pod2man(1)*, *nroff(1)*, *troff(1)*, *man(1)*, *man(7)*

Ossanna, Joseph F., and Brian W. Kernighan. "Troff User's Manual," Computing Science Technical Report No. 54, AT&T Bell Laboratories. This is the best documentation of standard **nroff** and **troff**. At the time of this writing, it's available at <http://www.cs.bell-labs.com/cm/cs/cstr.html>.

The man page documenting the man macro set may be *man(5)* instead of *man(7)* on your system. Also, please see *pod2man(1)* for extensive documentation on writing manual pages if you've not done it before and aren't familiar with the conventions.

The current version of this module is always available from its web site at

---

<http://www.eyrie.org/~eagle/software/podlators/>. It is also part of the Perl core distribution as of 5.6.0.

**AUTHOR**

Russ Allbery <rra@stanford.edu>, based *very* heavily on the original **pod2man** by Tom Christiansen <tchrist@mox.perl.com>.

**COPYRIGHT AND LICENSE**

Copyright 1999, 2000, 2001, 2002 by Russ Allbery <rra@stanford.edu>.

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

**NAME**

Pod::ParseLink – Parse an L<> formatting code in POD text

**SYNOPSIS**

```
use Pod::ParseLink;
my ($text, $inferred, $name, $section, $type) = parselink ($link);
```

**DESCRIPTION**

This module only provides a single function, `parselink()`, which takes the text of an L<> formatting code and parses it. It returns the anchor text for the link (if any was given), the anchor text possibly inferred from the name and section, the name or URL, the section if any, and the type of link. The type will be one of 'url', 'pod', or 'man', indicating a URL, a link to a POD page, or a link to a Unix manual page.

Parsing is implemented per [perlpodspec](#). For backward compatibility, links where there is no section and name contains spaces, or links where the entirety of the link (except for the anchor text if given) is enclosed in double-quotes are interpreted as links to a section (L</section>).

The inferred anchor text is implemented per [perlpodspec](#):

```
L<name> => L<name|name>
L</section> => L<"section"|/section>
L<name/section> => L<"section" in name|name/section>
```

The name may contain embedded E<> and Z<> formatting codes, and the section, anchor text, and inferred anchor text may contain any formatting codes. Any double quotes around the section are removed as part of the parsing, as is any leading or trailing whitespace.

If the text of the L<> escape is entirely enclosed in double quotes, it's interpreted as a link to a section for backwards compatibility.

No attempt is made to resolve formatting codes. This must be done after calling `parselink` (since E<> formatting codes can be used to escape characters that would otherwise be significant to the parser and resolving them before parsing would result in an incorrect parse of a formatting code like:

```
L<verticalE<verbar>barE<sol>slash>
```

which should be interpreted as a link to the `vertical|bar/slash` POD page and not as a link to the `slash` section of the `bar` POD page with an anchor text of `vertical`. Note that not only the anchor text will need to have formatting codes expanded, but so will the target of the link (to deal with E<> and Z<> formatting codes), and special handling of the section may be necessary depending on whether the translator wants to consider markup in sections to be significant when resolving links. See [perlpodspec](#) for more information.

**SEE ALSO**

[Pod::Parser](#)

The current version of this module is always available from its web site at <http://www.eyrie.org/~eagle/software/podlators/>.

**AUTHOR**

Russ Allbery <[rra@stanford.edu](mailto:rra@stanford.edu)>.

**COPYRIGHT AND LICENSE**

Copyright 2001 by Russ Allbery <[rra@stanford.edu](mailto:rra@stanford.edu)>.

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

**NAME**

Pod::Parser – base class for creating POD filters and translators

**SYNOPSIS**

```
use Pod::Parser;

package MyParser;
@ISA = qw(Pod::Parser);

sub command {
 my ($parser, $command, $paragraph, $line_num) = @_;
 ## Interpret the command and its text; sample actions might be:
 if ($command eq 'head1') { ... }
 elsif ($command eq 'head2') { ... }
 ## ... other commands and their actions
 my $out_fh = $parser->output_handle();
 my $expansion = $parser->interpolate($paragraph, $line_num);
 print $out_fh $expansion;
}

sub verbatim {
 my ($parser, $paragraph, $line_num) = @_;
 ## Format verbatim paragraph; sample actions might be:
 my $out_fh = $parser->output_handle();
 print $out_fh $paragraph;
}

sub textblock {
 my ($parser, $paragraph, $line_num) = @_;
 ## Translate/Format this block of text; sample actions might be:
 my $out_fh = $parser->output_handle();
 my $expansion = $parser->interpolate($paragraph, $line_num);
 print $out_fh $expansion;
}

sub interior_sequence {
 my ($parser, $seq_command, $seq_argument) = @_;
 ## Expand an interior sequence; sample actions might be:
 return "$seq_argument" if ($seq_command eq 'B');
 return "`$seq_argument'" if ($seq_command eq 'C');
 return "_${seq_argument}_" if ($seq_command eq 'I');
 ## ... other sequence commands and their resulting text
}

package main;

Create a parser object and have it parse file whose name was
given on the command-line (use STDIN if no files were given).
$parser = new MyParser();
$parser->parse_from_filehandle(*STDIN) if (@ARGV == 0);
for (@ARGV) { $parser->parse_from_file($_); }
```

**REQUIRES**

perl5.005, Pod::InputObjects, Exporter, Symbol, Carp

## EXPORTS

Nothing.

## DESCRIPTION

**Pod::Parser** is a base class for creating POD filters and translators. It handles most of the effort involved with parsing the POD sections from an input stream, leaving subclasses free to be concerned only with performing the actual translation of text.

**Pod::Parser** parses PODs, and makes method calls to handle the various components of the POD. Subclasses of **Pod::Parser** override these methods to translate the POD into whatever output format they desire.

## QUICK OVERVIEW

To create a POD filter for translating POD documentation into some other format, you create a subclass of **Pod::Parser** which typically overrides just the base class implementation for the following methods:

- **command()**
- **verbatim()**
- **textblock()**
- **interior\_sequence()**

You may also want to override the **begin\_input()** and **end\_input()** methods for your subclass (to perform any needed per-file and/or per-document initialization or cleanup).

If you need to perform any preprocessing of input before it is parsed you may want to override one or more of **preprocess\_line()** and/or **preprocess\_paragraph()**.

Sometimes it may be necessary to make more than one pass over the input files. If this is the case you have several options. You can make the first pass using **Pod::Parser** and override your methods to store the intermediate results in memory somewhere for the **end\_pod()** method to process. You could use **Pod::Parser** for several passes with an appropriate state variable to control the operation for each pass. If your input source can't be reset to start at the beginning, you can store it in some other structure as a string or an array and have that structure implement a **getline()** method (which is all that **parse\_from\_filehandle()** uses to read input).

Feel free to add any member data fields you need to keep track of things like current font, indentation, horizontal or vertical position, or whatever else you like. Be sure to read *"PRIVATE METHODS AND DATA"* to avoid name collisions.

For the most part, the **Pod::Parser** base class should be able to do most of the input parsing for you and leave you free to worry about how to interpret the commands and translate the result.

Note that all we have described here in this quick overview is the simplest most straightforward use of **Pod::Parser** to do stream-based parsing. It is also possible to use the **Pod::Parser::parse\_text** function to do more sophisticated tree-based parsing. See *"TREE-BASED PARSING"*.

## PARSING OPTIONS

A *parse-option* is simply a named option of **Pod::Parser** with a value that corresponds to a certain specified behavior. These various behaviors of **Pod::Parser** may be enabled/disabled by setting or unsetting one or more *parse-options* using the **parseopts()** method. The set of currently accepted parse-options is as follows:

### **-want\_nonPODs** (default: unset)

Normally (by default) **Pod::Parser** will only provide access to the POD sections of the input. Input paragraphs that are not part of the POD-format documentation are not made available to the caller (not even using **preprocess\_paragraph()**). Setting this option to a non-empty, non-zero value will allow **preprocess\_paragraph()** to see non-POD sections of the input as well as POD sections. The **cutting()** method can be used to determine if the corresponding paragraph is a POD paragraph,

or some other input paragraph.

#### **-process\_cut\_cmd** (default: unset)

Normally (by default) **Pod::Parser** handles the `=cut` POD directive by itself and does not pass it on to the caller for processing. Setting this option to a non-empty, non-zero value will cause **Pod::Parser** to pass the `=cut` directive to the caller just like any other POD command (and hence it may be processed by the `command()` method).

**Pod::Parser** will still interpret the `=cut` directive to mean that "cutting mode" has been (re)entered, but the caller will get a chance to capture the actual `=cut` paragraph itself for whatever purpose it desires.

#### **-warnings** (default: unset)

Normally (by default) **Pod::Parser** recognizes a bare minimum of pod syntax errors and warnings and issues diagnostic messages for errors, but not for warnings. (Use **Pod::Checker** to do more thorough checking of POD syntax.) Setting this option to a non-empty, non-zero value will cause **Pod::Parser** to issue diagnostics for the few warnings it recognizes as well as the errors.

Please see `"parseopts()"` for a complete description of the interface for the setting and unsetting of parse-options.

### RECOMMENDED SUBROUTINE/METHOD OVERRIDES

**Pod::Parser** provides several methods which most subclasses will probably want to override. These methods are as follows:

#### **command()**

```
$parser->command($cmd,$text,$line_num,$pod_para);
```

This method should be overridden by subclasses to take the appropriate action when a POD command paragraph (denoted by a line beginning with "=") is encountered. When such a POD directive is seen in the input, this method is called and is passed:

`$cmd`

the name of the command for this POD paragraph

`$text`

the paragraph text for the given POD paragraph command.

`$line_num`

the line-number of the beginning of the paragraph

`$pod_para`

a reference to a `Pod::Paragraph` object which contains further information about the paragraph command (see *Pod::InputObjects* for details).

**Note** that this method is called for `=pod` paragraphs.

The base class implementation of this method simply treats the raw POD command as normal block of paragraph text (invoking the `textblock()` method with the command paragraph).

#### **verbatim()**

```
$parser->verbatim($text,$line_num,$pod_para);
```

This method may be overridden by subclasses to take the appropriate action when a block of verbatim text is encountered. It is passed the following parameters:

`$text`

the block of text for the verbatim paragraph

`$line_num`

the line-number of the beginning of the paragraph

**\$pod\_para**

a reference to a `Pod::Paragraph` object which contains further information about the paragraph (see [Pod::InputObjects](#) for details).

The base class implementation of this method simply prints the textblock (unmodified) to the output filehandle.

**textblock()**

```
$parser->textblock($text,$line_num,$pod_para);
```

This method may be overridden by subclasses to take the appropriate action when a normal block of POD text is encountered (although the base class method will usually do what you want). It is passed the following parameters:

**\$text**

the block of text for the a POD paragraph

**\$line\_num**

the line-number of the beginning of the paragraph

**\$pod\_para**

a reference to a `Pod::Paragraph` object which contains further information about the paragraph (see [Pod::InputObjects](#) for details).

In order to process interior sequences, subclasses implementations of this method will probably want to invoke either **interpolate()** or **parse\_text()**, passing it the text block `$text`, and the corresponding line number in `$line_num`, and then perform any desired processing upon the returned result.

The base class implementation of this method simply prints the text block as it occurred in the input stream).

**interior\_sequence()**

```
$parser->interior_sequence($seq_cmd,$seq_arg,$pod_seq);
```

This method should be overridden by subclasses to take the appropriate action when an interior sequence is encountered. An interior sequence is an embedded command within a block of text which appears as a command name (usually a single uppercase character) followed immediately by a string of text which is enclosed in angle brackets. This method is passed the sequence command `$seq_cmd` and the corresponding text `$seq_arg`. It is invoked by the **interpolate()** method for each interior sequence that occurs in the string that it is passed. It should return the desired text string to be used in place of the interior sequence. The `$pod_seq` argument is a reference to a `Pod::InteriorSequence` object which contains further information about the interior sequence. Please see [Pod::InputObjects](#) for details if you need to access this additional information.

Subclass implementations of this method may wish to invoke the **nested()** method of `$pod_seq` to see if it is nested inside some other interior-sequence (and if so, which kind).

The base class implementation of the **interior\_sequence()** method simply returns the raw text of the interior sequence (as it occurred in the input) to the caller.

**OPTIONAL SUBROUTINE/METHOD OVERRIDES**

**Pod::Parser** provides several methods which subclasses may want to override to perform any special pre/post-processing. These methods do *not* have to be overridden, but it may be useful for subclasses to take advantage of them.

**new()**

```
my $parser = Pod::Parser->new();
```

This is the constructor for **Pod::Parser** and its subclasses. You *do not* need to override this method! It is capable of constructing subclass objects as well as base class objects, provided you use any of the following



constructor invocation styles:

```
my $parser1 = MyParser->new();
my $parser2 = new MyParser();
my $parser3 = $parser2->new();
```

where `MyParser` is some subclass of **Pod::Parser**.

Using the syntax `MyParser::new()` to invoke the constructor is *not* recommended, but if you insist on being able to do this, then the subclass *will* need to override the **new()** constructor method. If you do override the constructor, you *must* be sure to invoke the **initialize()** method of the newly blessed object.

Using any of the above invocations, the first argument to the constructor is always the corresponding package name (or object reference). No other arguments are required, but if desired, an associative array (or hash-table) may be passed to the **new()** constructor, as in:

```
my $parser1 = MyParser->new(MYDATA => $value1, MOREDATA => $value2);
my $parser2 = new MyParser(-myflag => 1);
```

All arguments passed to the **new()** constructor will be treated as key/value pairs in a hash-table. The newly constructed object will be initialized by copying the contents of the given hash-table (which may have been empty). The **new()** constructor for this class and all of its subclasses returns a blessed reference to the initialized object (hash-table).

#### **initialize()**

```
$parser->initialize();
```

This method performs any necessary object initialization. It takes no arguments (other than the object instance of course, which is typically copied to a local variable named `$self`). If subclasses override this method then they *must* be sure to invoke `$self->SUPER::initialize()`.

#### **begin\_pod()**

```
$parser->begin_pod();
```

This method is invoked at the beginning of processing for each POD document that is encountered in the input. Subclasses should override this method to perform any per-document initialization.

#### **begin\_input()**

```
$parser->begin_input();
```

This method is invoked by **parse\_from\_filehandle()** immediately *before* processing input from a filehandle. The base class implementation does nothing, however, subclasses may override it to perform any per-file initializations.

Note that if multiple files are parsed for a single POD document (perhaps the result of some future `=include` directive) this method is invoked for every file that is parsed. If you wish to perform certain initializations once per document, then you should use **begin\_pod()**.

#### **end\_input()**

```
$parser->end_input();
```

This method is invoked by **parse\_from\_filehandle()** immediately *after* processing input from a filehandle. The base class implementation does nothing, however, subclasses may override it to perform any per-file cleanup actions.

Please note that if multiple files are parsed for a single POD document (perhaps the result of some kind of `=include` directive) this method is invoked for every file that is parsed. If you wish to perform certain cleanup actions once per document, then you should use **end\_pod()**.

**end\_pod()**

```
$parser->end_pod();
```

This method is invoked at the end of processing for each POD document that is encountered in the input. Subclasses should override this method to perform any per-document finalization.

**preprocess\_line()**

```
$textline = $parser->preprocess_line($text, $line_num);
```

This method should be overridden by subclasses that wish to perform any kind of preprocessing for each *line* of input (*before* it has been determined whether or not it is part of a POD paragraph). The parameter `$text` is the input line; and the parameter `$line_num` is the line number of the corresponding text line.

The value returned should correspond to the new text to use in its place. If the empty string or an undefined value is returned then no further processing will be performed for this line.

Please note that the **preprocess\_line()** method is invoked *before* the **preprocess\_paragraph()** method. After all (possibly preprocessed) lines in a paragraph have been assembled together and it has been determined that the paragraph is part of the POD documentation from one of the selected sections, then **preprocess\_paragraph()** is invoked.

The base class implementation of this method returns the given text.

**preprocess\_paragraph()**

```
$textblock = $parser->preprocess_paragraph($text, $line_num);
```

This method should be overridden by subclasses that wish to perform any kind of preprocessing for each block (paragraph) of POD documentation that appears in the input stream. The parameter `$text` is the POD paragraph from the input file; and the parameter `$line_num` is the line number for the beginning of the corresponding paragraph.

The value returned should correspond to the new text to use in its place. If the empty string is returned or an undefined value is returned, then the given `$text` is ignored (not processed).

This method is invoked after gathering up all the lines in a paragraph and after determining the cutting state of the paragraph, but before trying to further parse or interpret them. After **preprocess\_paragraph()** returns, the current cutting state (which is returned by `$self->cutting()`) is examined. If it evaluates to true then input text (including the given `$text`) is cut (not processed) until the next POD directive is encountered.

Please note that the **preprocess\_line()** method is invoked *before* the **preprocess\_paragraph()** method. After all (possibly preprocessed) lines in a paragraph have been assembled together and either it has been determined that the paragraph is part of the POD documentation from one of the selected sections or the `-want_nonPODs` option is true, then **preprocess\_paragraph()** is invoked.

The base class implementation of this method returns the given text.

**METHODS FOR PARSING AND PROCESSING**

**Pod::Parser** provides several methods to process input text. These methods typically won't need to be overridden (and in some cases they can't be overridden), but subclasses may want to invoke them to exploit their functionality.

**parse\_text()**

```
$ptree1 = $parser->parse_text($text, $line_num);
$ptree2 = $parser->parse_text({%opts}, $text, $line_num);
$ptree3 = $parser->parse_text(\%opts, $text, $line_num);
```

This method is useful if you need to perform your own interpolation of interior sequences and can't rely upon **interpolate** to expand them in simple bottom-up order.

The parameter `$text` is a string or block of text to be parsed for interior sequences; and the parameter

`$line_num` is the line number corresponding to the beginning of `$text`.

**parse\_text()** will parse the given text into a parse-tree of "nodes." and interior-sequences. Each "node" in the parse tree is either a text-string, or a **Pod::InteriorSequence**. The result returned is a parse-tree of type **Pod::ParseTree**. Please see [Pod::InputObjects](#) for more information about **Pod::InteriorSequence** and **Pod::ParseTree**.

If desired, an optional hash-ref may be specified as the first argument to customize certain aspects of the parse-tree that is created and returned. The set of recognized option keywords are:

**-expand\_seq => code-ref|method-name**

Normally, the parse-tree returned by **parse\_text()** will contain an unexpanded **Pod::InteriorSequence** object for each interior-sequence encountered. Specifying **-expand\_seq** tells **parse\_text()** to "expand" every interior-sequence it sees by invoking the referenced function (or named method of the parser object) and using the return value as the expanded result.

If a subroutine reference was given, it is invoked as:

```
&$code_ref($parser, $sequence)
```

and if a method-name was given, it is invoked as:

```
$parser->method_name($sequence)
```

where `$parser` is a reference to the parser object, and `$sequence` is a reference to the interior-sequence object. [NOTE: If the **interior\_sequence()** method is specified, then it is invoked according to the interface specified in "[interior\\_sequence\(\)](#)".].

**-expand\_text => code-ref|method-name**

Normally, the parse-tree returned by **parse\_text()** will contain a text-string for each contiguous sequence of characters outside of an interior-sequence. Specifying **-expand\_text** tells **parse\_text()** to "preprocess" every such text-string it sees by invoking the referenced function (or named method of the parser object) and using the return value as the preprocessed (or "expanded") result. [Note that if the result is an interior-sequence, then it will *not* be expanded as specified by the **-expand\_seq** option; Any such recursive expansion needs to be handled by the specified callback routine.]

If a subroutine reference was given, it is invoked as:

```
&$code_ref($parser, $text, $ptree_node)
```

and if a method-name was given, it is invoked as:

```
$parser->method_name($text, $ptree_node)
```

where `$parser` is a reference to the parser object, `$text` is the text-string encountered, and `$ptree_node` is a reference to the current node in the parse-tree (usually an interior-sequence object or else the top-level node of the parse-tree).

**-expand\_ptree => code-ref|method-name**

Rather than returning a **Pod::ParseTree**, pass the parse-tree as an argument to the referenced subroutine (or named method of the parser object) and return the result instead of the parse-tree object.

If a subroutine reference was given, it is invoked as:

```
&$code_ref($parser, $ptree)
```

and if a method-name was given, it is invoked as:

```
$parser->method_name($ptree)
```

where `$parser` is a reference to the parser object, and `$ptree` is a reference to the parse-tree object.

**interpolate()**

```
$textblock = $parser->interpolate($text, $line_num);
```

This method translates all text (including any embedded interior sequences) in the given text string `$text` and returns the interpolated result. The parameter `$line_num` is the line number corresponding to the beginning of `$text`.

**interpolate()** merely invokes a private method to recursively expand nested interior sequences in bottom-up order (innermost sequences are expanded first). If there is a need to expand nested sequences in some alternate order, use **parse\_text** instead.

**parse\_paragraph()**

```
$parser->parse_paragraph($text, $line_num);
```

This method takes the text of a POD paragraph to be processed, along with its corresponding line number, and invokes the appropriate method (one of **command()**, **verbatim()**, or **textblock()**).

For performance reasons, this method is invoked directly without any dynamic lookup; Hence subclasses may *not* override it!

```
=end __PRIVATE__
```

**parse\_from\_filehandle()**

```
$parser->parse_from_filehandle($in_fh,$out_fh);
```

This method takes an input filehandle (which is assumed to already be opened for reading) and reads the entire input stream looking for blocks (paragraphs) of POD documentation to be processed. If no first argument is given the default input filehandle STDIN is used.

The `$in_fh` parameter may be any object that provides a **getline()** method to retrieve a single line of input text (hence, an appropriate wrapper object could be used to parse PODs from a single string or an array of strings).

Using `$in_fh->getline()`, input is read line-by-line and assembled into paragraphs or "blocks" (which are separated by lines containing nothing but whitespace). For each block of POD documentation encountered it will invoke a method to parse the given paragraph.

If a second argument is given then it should correspond to a filehandle where output should be sent (otherwise the default output filehandle is STDOUT if no output filehandle is currently in use).

**NOTE:** For performance reasons, this method caches the input stream at the top of the stack in a local variable. Any attempts by clients to change the stack contents during processing when in the midst executing of this method *will not affect* the input stream used by the current invocation of this method.

This method does *not* usually need to be overridden by subclasses.

**parse\_from\_file()**

```
$parser->parse_from_file($filename,$outfile);
```

This method takes a filename and does the following:

- opens the input and output files for reading (creating the appropriate filehandles)
- invokes the **parse\_from\_filehandle()** method passing it the corresponding input and output filehandles.
- closes the input and output files.

If the special input filename `"-"` or `"<&STDIN"` is given then the STDIN filehandle is used for input (and no open or close is performed). If no input filename is specified then `"-"` is implied.

If a second argument is given then it should be the name of the desired output file. If the special output filename `"-"` or `"&STDOUT"` is given then the STDOUT filehandle is used for output (and no open or close

is performed). If the special output filename "&STDERR" is given then the STDERR filehandle is used for output (and no open or close is performed). If no output filehandle is currently in use and no output filename is specified, then "-" is implied.

This method does *not* usually need to be overridden by subclasses.

## ACCESSOR METHODS

Clients of **Pod::Parser** should use the following methods to access instance data fields:

### errorsub()

```
$parser->errorsub("method_name");
$parser->errorsub(\&warn_user);
$parser->errorsub(sub { print STDERR, @_ });
```

Specifies the method or subroutine to use when printing error messages about POD syntax. The supplied method/subroutine *must* return TRUE upon successful printing of the message. If undef is given, then the **warn** builtin is used to issue error messages (this is the default behavior).

```
my $errorsub = $parser->errorsub();
my $errmsg = "This is an error message!\n"
(ref $errorsub) and &{$errorsub}($errmsg)
 or (defined $errorsub) and $parser->$errorsub($errmsg)
 or warn($errmsg);
```

Returns a method name, or else a reference to the user-supplied subroutine used to print error messages. Returns undef if the **warn** builtin is used to issue error messages (this is the default behavior).

### cutting()

```
$boolean = $parser->cutting();
```

Returns the current cutting state: a boolean-valued scalar which evaluates to true if text from the input file is currently being "cut" (meaning it is *not* considered part of the POD document).

```
$parser->cutting($boolean);
```

Sets the current cutting state to the given value and returns the result.

### parseopts()

When invoked with no additional arguments, **parseopts** returns a hashtable of all the current parsing options.

```
See if we are parsing non-POD sections as well as POD ones
my %opts = $parser->parseopts();
$opts{'-want_nonPODs'} and print "-want_nonPODs\n";
```

When invoked using a single string, **parseopts** treats the string as the name of a parse-option and returns its corresponding value if it exists (returns undef if it doesn't).

```
Did we ask to see '=cut' paragraphs?
my $want_cut = $parser->parseopts('-process_cut_cmd');
$want_cut and print "-process_cut_cmd\n";
```

When invoked with multiple arguments, **parseopts** treats them as key/value pairs and the specified parse-option names are set to the given values. Any unspecified parse-options are unaffected.

```
Set them back to the default
$parser->parseopts(-warnings => 0);
```

When passed a single hash-ref, **parseopts** uses that hash to completely reset the existing parse-options, all previous parse-option values are lost.

```
Reset all options to default
$parser->parseopts({ });
```

See *"PARSING OPTIONS"* for more information on the name and meaning of each parse-option currently recognized.

#### **output\_file()**

```
$fname = $parser->output_file();
```

Returns the name of the output file being written.

#### **output\_handle()**

```
$fhandle = $parser->output_handle();
```

Returns the output filehandle object.

#### **input\_file()**

```
$fname = $parser->input_file();
```

Returns the name of the input file being read.

#### **input\_handle()**

```
$fhandle = $parser->input_handle();
```

Returns the current input filehandle object.

#### **input\_streams()**

```
$listref = $parser->input_streams();
```

Returns a reference to an array which corresponds to the stack of all the input streams that are currently in the middle of being parsed.

While parsing an input stream, it is possible to invoke **parse\_from\_file()** or **parse\_from\_filehandle()** to parse a new input stream and then return to parsing the previous input stream. Each input stream to be parsed is pushed onto the end of this input stack before any of its input is read. The input stream that is currently being parsed is always at the end (or top) of the input stack. When an input stream has been exhausted, it is popped off the end of the input stack.

Each element on this input stack is a reference to `Pod::InputSource` object. Please see [Pod::InputObjects](#) for more details.

This method might be invoked when printing diagnostic messages, for example, to obtain the name and line number of the all input files that are currently being processed.

```
=end __PRIVATE__
```

#### **top\_stream()**

```
$hashref = $parser->top_stream();
```

Returns a reference to the hash-table that represents the element that is currently at the top (end) of the input stream stack (see *"input\_streams()"*). The return value will be the `undef` if the input stack is empty.

This method might be used when printing diagnostic messages, for example, to obtain the name and line number of the current input file.

```
=end __PRIVATE__
```

### **PRIVATE METHODS AND DATA**

**Pod::Parser** makes use of several internal methods and data fields which clients should not need to see or use. For the sake of avoiding name collisions for client data and methods, these methods and fields are briefly discussed here. Determined hackers may obtain further information about them by reading the **Pod::Parser** source code.

Private data fields are stored in the hash-object whose reference is returned by the **new()** constructor for this class. The names of all private methods and data-fields used by **Pod::Parser** begin with a prefix of `"_"` and match the regular expression `/^_\w+$/`.

**`_push_input_stream()`**

```
$hashref = $parser->_push_input_stream($in_fh,$out_fh);
```

This method will push the given input stream on the input stack and perform any necessary beginning-of-document or beginning-of-file processing. The argument `$in_fh` is the input stream filehandle to push, and `$out_fh` is the corresponding output filehandle to use (if it is not given or is undefined, then the current output stream is used, which defaults to standard output if it doesn't exist yet).

The value returned will be reference to the hash-table that represents the new top of the input stream stack. *Please Note* that it is possible for this method to use default values for the input and output file handles. If this happens, you will need to look at the `INPUT` and `OUTPUT` instance data members to determine their new values.

```
=end_PRIVATE_
```

**`_pop_input_stream()`**

```
$hashref = $parser->_pop_input_stream();
```

This takes no arguments. It will perform any necessary end-of-file or end-of-document processing and then pop the current input stream from the top of the input stack.

The value returned will be reference to the hash-table that represents the new top of the input stream stack.

```
=end_PRIVATE_
```

**TREE-BASED PARSING**

If straightforward stream-based parsing won't meet your needs (as is likely the case for tasks such as translating PODs into structured markup languages like HTML and XML) then you may need to take the tree-based approach. Rather than doing everything in one pass and calling the `interpolate()` method to expand sequences into text, it may be desirable to instead create a parse-tree using the `parse_text()` method to return a tree-like structure which may contain an ordered list of children (each of which may be a text-string, or a similar tree-like structure).

Pay special attention to *"METHODS FOR PARSING AND PROCESSING"* and to the objects described in *Pod::InputObjects*. The former describes the gory details and parameters for how to customize and extend the parsing behavior of **Pod::Parser**. **Pod::InputObjects** provides several objects that may all be used interchangeably as parse-trees. The most obvious one is the **Pod::ParseTree** object. It defines the basic interface and functionality that all things trying to be a POD parse-tree should do. A **Pod::ParseTree** is defined such that each "node" may be a text-string, or a reference to another parse-tree. Each **Pod::Paragraph** object and each **Pod::InteriorSequence** object also supports the basic parse-tree interface.

The `parse_text()` method takes a given paragraph of text, and returns a parse-tree that contains one or more children, each of which may be a text-string, or an `InteriorSequence` object. There are also callback-options that may be passed to `parse_text()` to customize the way it expands or transforms interior-sequences, as well as the returned result. These callbacks can be used to create a parse-tree with custom-made objects (which may or may not support the parse-tree interface, depending on how you choose to do it).

If you wish to turn an entire POD document into a parse-tree, that process is fairly straightforward. The `parse_text()` method is the key to doing this successfully. Every paragraph-callback (i.e. the polymorphic methods for `command()`, `verbatim()`, and `textblock()` paragraphs) takes a **Pod::Paragraph** object as an argument. Each paragraph object has a `parse_tree()` method that can be used to get or set a corresponding parse-tree. So for each of those paragraph-callback methods, simply call `parse_text()` with the options you desire, and then use the returned parse-tree to assign to the given paragraph object.

That gives you a parse-tree for each paragraph – so now all you need is an ordered list of paragraphs. You can maintain that yourself as a data element in the object/hash. The most straightforward way would be

simply to use an array-ref, with the desired set of custom "options" for each invocation of **parse\_text**. Let's assume the desired option-set is given by the hash %options. Then we might do something like the following:

```
package MyPodParserTree;

@ISA = qw(Pod::Parser);

...

sub begin_pod {
 my $self = shift;
 $self->{'-paragraphs'} = []; ## initialize paragraph list
}

sub command {
 my ($parser, $command, $paragraph, $line_num, $pod_para) = @_;
 my $ptree = $parser->parse_text(%options, $paragraph, ...);
 $pod_para->parse_tree($ptree);
 push @{ $self->{'-paragraphs'} }, $pod_para;
}

sub verbatim {
 my ($parser, $paragraph, $line_num, $pod_para) = @_;
 push @{ $self->{'-paragraphs'} }, $pod_para;
}

sub textblock {
 my ($parser, $paragraph, $line_num, $pod_para) = @_;
 my $ptree = $parser->parse_text(%options, $paragraph, ...);
 $pod_para->parse_tree($ptree);
 push @{ $self->{'-paragraphs'} }, $pod_para;
}

...

package main;

...
my $parser = new MyPodParserTree(...);
$parser->parse_from_file(...);
my $paragraphs_ref = $parser->{'-paragraphs'};
```

Of course, in this module-author's humble opinion, I'd be more inclined to use the existing **Pod::ParseTree** object than a simple array. That way everything in it, paragraphs and sequences, all respond to the same core interface for all parse-tree nodes. The result would look something like:

```
package MyPodParserTree2;

...

sub begin_pod {
 my $self = shift;
 $self->{'-ptree'} = new Pod::ParseTree; ## initialize parse-tree
}

sub parse_tree {
 ## convenience method to get/set the parse-tree for the entire POD
 (@_ > 1) and $_[0]->{'-ptree'} = $_[1];
 return $_[0]->{'-ptree'};
}
```



```

sub command {
 my ($parser, $command, $paragraph, $line_num, $pod_para) = @_;
 my $ptree = $parser->parse_text({<<options>>}, $paragraph, ...);
 $pod_para->parse_tree($ptree);
 $parser->parse_tree()->append($pod_para);
}

sub verbatim {
 my ($parser, $paragraph, $line_num, $pod_para) = @_;
 $parser->parse_tree()->append($pod_para);
}

sub textblock {
 my ($parser, $paragraph, $line_num, $pod_para) = @_;
 my $ptree = $parser->parse_text({<<options>>}, $paragraph, ...);
 $pod_para->parse_tree($ptree);
 $parser->parse_tree()->append($pod_para);
}

...

package main;
...
my $parser = new MyPodParserTree2(...);
$parser->parse_from_file(...);
my $ptree = $parser->parse_tree;
...

```

Now you have the entire POD document as one great big parse-tree. You can even use the `--expand_seq` option to `parse_text` to insert whole different kinds of objects. Just don't expect `Pod::Parser` to know what to do with them after that. That will need to be in your code. Or, alternatively, you can insert any object you like so long as it conforms to the `Pod::ParseTree` interface.

One could use this to create subclasses of `Pod::Paragraphs` and `Pod::InteriorSequences` for specific commands (or to create your own custom node-types in the parse-tree) and add some kind of `emit()` method to each custom node/subclass object in the tree. Then all you'd need to do is recursively walk the tree in the desired order, processing the children (most likely from left to right) by formatting them if they are text-strings, or by calling their `emit()` method if they are objects/references.

## SEE ALSO

[Pod::InputObjects](#), [Pod::Select](#)

**Pod::InputObjects** defines POD input objects corresponding to command paragraphs, parse-trees, and interior-sequences.

**Pod::Select** is a subclass of **Pod::Parser** which provides the ability to selectively include and/or exclude sections of a POD document from being translated based upon the current heading, subheading, subsubheading, etc.

=for \_\_PRIVATE\_\_ **Pod::Callbacks** is a subclass of **Pod::Parser** which gives its users the ability the employ *callback functions* instead of, or in addition to, overriding methods of the base class.

=for \_\_PRIVATE\_\_ **Pod::Select** and **Pod::Callbacks** do not override any methods nor do they define any new methods with the same name. Because of this, they may *both* be used (in combination) as a base class of the same subclass in order to combine their functionality without causing any namespace clashes due to multiple inheritance.

**AUTHOR**

Brad Appleton <bradapp@enteract.com>

Based on code for **Pod::Text** written by Tom Christiansen <tchrist@mox.perl.com>

## NAME

Pod::ParseUtils – helpers for POD parsing and conversion

## SYNOPSIS

```
use Pod::ParseUtils;

my $list = new Pod::List;
my $link = Pod::Hyperlink->new('Pod::Parser');
```

## DESCRIPTION

**Pod::ParseUtils** contains a few object-oriented helper packages for POD parsing and processing (i.e. in POD formatters and translators).

### Pod::List

**Pod::List** can be used to hold information about POD lists (written as `=over ... =item ... =back`) for further processing. The following methods are available:

`Pod::List->new()`

Create a new list object. Properties may be specified through a hash reference like this:

```
my $list = Pod::List->new({ -start => $., -indent => 4 });
```

See the individual methods/properties for details.

`$list->file()`

Without argument, retrieves the file name the list is in. This must have been set before by either specifying **-file** in the **new()** method or by calling the **file()** method with a scalar argument.

`$list->start()`

Without argument, retrieves the line number where the list started. This must have been set before by either specifying **-start** in the **new()** method or by calling the **start()** method with a scalar argument.

`$list->indent()`

Without argument, retrieves the indent level of the list as specified in `=over n`. This must have been set before by either specifying **-indent** in the **new()** method or by calling the **indent()** method with a scalar argument.

`$list->type()`

Without argument, retrieves the list type, which can be an arbitrary value, e.g. OL, UL, ... when thinking the HTML way. This must have been set before by either specifying **-type** in the **new()** method or by calling the **type()** method with a scalar argument.

`$list->rx()`

Without argument, retrieves a regular expression for simplifying the individual item strings once the list type has been determined. Usage: E.g. when converting to HTML, one might strip the leading number in an ordered list as `<OL>` already prints numbers itself. This must have been set before by either specifying **-rx** in the **new()** method or by calling the **rx()** method with a scalar argument.

`$list->item()`

Without argument, retrieves the array of the items in this list. The items may be represented by any scalar. If an argument has been given, it is pushed on the list of items.

`$list->parent()`

Without argument, retrieves information about the parent holding this list, which is represented as an arbitrary scalar. This must have been set before by either specifying **-parent** in the **new()** method or by calling the **parent()** method with a scalar argument.

**\$list->tag()**

Without argument, retrieves information about the list tag, which can be any scalar. This must have been set before by either specifying **-tag** in the **new()** method or by calling the **tag()** method with a scalar argument.

**Pod::Hyperlink**

**Pod::Hyperlink** is a class for manipulation of POD hyperlinks. Usage:

```
my $link = Pod::Hyperlink->new('alternative text|page/"section in page"');
```

The **Pod::Hyperlink** class is mainly designed to parse the contents of the `L<...>` sequence, providing a simple interface for accessing the different parts of a POD hyperlink for further processing. It can also be used to construct hyperlinks.

**Pod::Hyperlink->new()**

The **new()** method can either be passed a set of key/value pairs or a single scalar value, namely the contents of a `L<...>` sequence. An object of the class **Pod::Hyperlink** is returned. The value `undef` indicates a failure, the error message is stored in `$@`.

**\$link->parse(\$string)**

This method can be used to (re)parse a (new) hyperlink, i.e. the contents of a `L<...>` sequence. The result is stored in the current object. Warnings are stored in the **warnings** property. E.g. sections like `L<open(2)>` are deprecated, as they do not point to Perl documents. `L<DBI::foo(3p)>` is wrong as well, the manpage section can simply be dropped.

**\$link->markup(\$string)**

Set/retrieve the textual value of the link. This string contains special markers `P<>` and `Q<>` that should be expanded by the translator's interior sequence expansion engine to the formatter-specific code to highlight/activate the hyperlink. The details have to be implemented in the translator.

**\$link->text()**

This method returns the textual representation of the hyperlink as above, but without markers (read only). Depending on the link type this is one of the following alternatives (the `+` and `*` denote the portions of the text that are marked up):

```
the +perl+ manpage
the *$|* entry in the +perlvar+ manpage
the section on *OPTIONS* in the +perldoc+ manpage
the section on *DESCRIPTION* elsewhere in this document
```

**\$link->warning()**

After parsing, this method returns any warnings encountered during the parsing process.

**\$link->file()****\$link->line()**

Just simple slots for storing information about the line and the file the link was encountered in. Has to be filled in manually.

**\$link->page()**

This method sets or returns the POD page this link points to.

**\$link->node()**

As above, but the destination node text of the link.

**\$link->alttext()**

Sets or returns an alternative text specified in the link.

**\$link->type()**

The node type, either `section` or `item`. As an unofficial type, there is also `hyperlink`, derived from e.g. `L<http://perl.com>`

**\$link->link()**

Returns the link as contents of `L<>`. Reciprocal to **parse()**.

**Pod::Cache**

**Pod::Cache** holds information about a set of POD documents, especially the nodes for hyperlinks. The following methods are available:

**Pod::Cache->new()**

Create a new cache object. This object can hold an arbitrary number of POD documents of class **Pod::Cache::Item**.

**\$cache->item()**

Add a new item to the cache. Without arguments, this method returns a list of all cache elements.

**\$cache->find\_page(\$name)**

Look for a POD document named `$name` in the cache. Returns the reference to the corresponding **Pod::Cache::Item** object or `undef` if not found.

**Pod::Cache::Item**

**Pod::Cache::Item** holds information about individual POD documents, that can be grouped in a **Pod::Cache** object. It is intended to hold information about the hyperlink nodes of POD documents. The following methods are available:

**Pod::Cache::Item->new()**

Create a new object.

**\$cacheitem->page()**

Set/retrieve the POD document name (e.g. `"Pod::Parser"`).

**\$cacheitem->description()**

Set/retrieve the POD short description as found in the `=head1 NAME` section.

**\$cacheitem->path()**

Set/retrieve the POD file storage path.

**\$cacheitem->file()**

Set/retrieve the POD file name.

**\$cacheitem->nodes()**

Add a node (or a list of nodes) to the document's node list. Note that the order is kept, i.e. start with the first node and end with the last. If no argument is given, the current list of nodes is returned in the same order the nodes have been added. A node can be any scalar, but usually is a pair of node string and unique id for the `find_node` method to work correctly.

**\$cacheitem->find\_node(\$name)**

Look for a node or index entry named `$name` in the object. Returns the unique id of the node (i.e. the second element of the array stored in the node array) or `undef` if not found.

**\$cacheitem->idx()**

Add an index entry (or a list of them) to the document's index list. Note that the order is kept, i.e. start with the first node and end with the last. If no argument is given, the current list of index entries is returned in the same order the entries have been added. An index entry can be any scalar, but usually is a pair of string and unique id.

**AUTHOR**

Marek Rouchal <marek@saftsack.fs.uni-bayreuth.de>, borrowing a lot of things from [pod2man](#) and [pod2roff](#) as well as other POD processing tools by Tom Christiansen, Brad Appleton and Russ Allbery.

**SEE ALSO**

[pod2man](#), [pod2roff](#), [Pod::Parser](#), [Pod::Checker](#), [pod2html](#)

**NAME**

Pod::Plainer – Perl extension for converting Pod to old style Pod.

**SYNOPSIS**

```
use Pod::Plainer;

my $parser = Pod::Plainer -> new ();
$parser -> parse_from_filehandle(*STDIN);
```

**DESCRIPTION**

Pod::Plainer uses Pod::Parser which takes Pod with the (new) ‘C<< .. >>’ constructs and returns the old(er) style with just ‘C<>’; ‘<’ and ‘>’ are replaced by ‘E<It>’ and ‘E<gt>’.

This can be used to pre-process Pod before using tools which do not recognise the new style Pods.

**EXPORT**

None by default.

**AUTHOR**

Robin Barker, rmb1@cise.npl.co.uk

**SEE ALSO**

See [Pod::Parser](#).

**NAME**

Pod::Select, podselect() – extract selected sections of POD from input

**SYNOPSIS**

```
use Pod::Select;

Select all the POD sections for each file in @filelist
and print the result on standard output.
podselect(@filelist);

Same as above, but write to tmp.out
podselect({-output => "tmp.out"}, @filelist);

Select from the given filelist, only those POD sections that are
within a 1st level section named any of: NAME, SYNOPSIS, OPTIONS.
podselect({-sections => ["NAME|SYNOPSIS", "OPTIONS"]}, @filelist);

Select the "DESCRIPTION" section of the PODs from STDIN and write
the result to STDERR.
podselect({-output => ">&STDERR", -sections => ["DESCRIPTION"]}, *STDIN);

or

use Pod::Select;

Create a parser object for selecting POD sections from the input
$parser = new Pod::Select();

Select all the POD sections for each file in @filelist
and print the result to tmp.out.
$parser->parse_from_file("<&STDIN", "tmp.out");

Select from the given filelist, only those POD sections that are
within a 1st level section named any of: NAME, SYNOPSIS, OPTIONS.
$parser->select("NAME|SYNOPSIS", "OPTIONS");
for (@filelist) { $parser->parse_from_file($_); }

Select the "DESCRIPTION" and "SEE ALSO" sections of the PODs from
STDIN and write the result to STDERR.
$parser->select("DESCRIPTION");
$parser->add_selection("SEE ALSO");
$parser->parse_from_filehandle(*STDIN, *STDERR);
```

**REQUIRES**

perl5.005, Pod::Parser, Exporter, Carp

**EXPORTS**

podselect()

**DESCRIPTION**

**podselect()** is a function which will extract specified sections of pod documentation from an input stream. This ability is provided by the **Pod::Select** module which is a subclass of **Pod::Parser**. **Pod::Select** provides a method named **select()** to specify the set of POD sections to select for processing/printing. **podselect()** merely creates a **Pod::Select** object and then invokes the **podselect()** followed by **parse\_from\_file()**.

**SECTION SPECIFICATIONS**

**podselect()** and **Pod::Select::select()** may be given one or more "section specifications" to restrict the text processed to only the desired set of sections and their corresponding subsections. A section specification is a string containing one or more Perl-style regular expressions separated by forward slashes



(""). If you need to use a forward slash literally within a section title you can escape it with a backslash ("\\").

The formal syntax of a section specification is:

- *head1–title–regex/head2–title–regex/...*

Any omitted or empty regular expressions will default to ".\*". Please note that each regular expression given is implicitly anchored by adding "^" and "\$" to the beginning and end. Also, if a given regular expression starts with a "!" character, then the expression is *negated* (so `!foo` would match anything *except* `foo`).

Some example section specifications follow.

- Match the NAME and SYNOPSIS sections and all of their subsections:

NAME | SYNOPSIS

- Match only the Question and Answer subsections of the DESCRIPTION section:

DESCRIPTION/Question | Answer

- Match the Comments subsection of *all* sections:

/Comments

- Match all subsections of DESCRIPTION *except* for Comments:

DESCRIPTION/!Comments

- Match the DESCRIPTION section but do *not* match any of its subsections:

DESCRIPTION/!.+

- Match all top level sections but none of their subsections:

/!.+

=begin \_NOT\_IMPLEMENTED\_

## RANGE SPECIFICATIONS

`podselect()` and `Pod::Select::select()` may be given one or more "range specifications" to restrict the text processed to only the desired ranges of paragraphs in the desired set of sections. A range specification is a string containing a single Perl-style regular expression (a regex), or else two Perl-style regular expressions (regexs) separated by a ".." (Perl's "range" operator is ".."). The regexs in a range specification are delimited by forward slashes ("/"). If you need to use a forward slash literally within a regex you can escape it with a backslash ("\\").

The formal syntax of a range specification is:

- */start–range–regex[/end–range–regex/]*

Where each the item inside square brackets (the "." followed by the end-range-regex) is optional. Each "range-regex" is of the form:

=cmd–expr text–expr

Where *cmd–expr* is intended to match the name of one or more POD commands, and *text–expr* is intended to match the paragraph text for the command. If a range-regex is supposed to match a POD command, then the first character of the regex (the one after the initial '/') absolutely *must* be a single '=' character; it may not be anything else (not even a regex meta-character) if it is supposed to match against the name of a POD command.

If no *=cmd–expr* is given then the *text–expr* will be matched against plain textblocks unless it is preceded by a space, in which case it is matched against verbatim text–blocks. If no *text–expr* is given then only the command–portion of the paragraph is matched against.

Note that these two expressions are each implicitly anchored. This means that when matching against the `command-name`, there will be an implicit `^` and `$` around the given `=cmd-expr`; and when matching against the paragraph text there will be an implicit `\A` and `\Z` around the given `text-expr`.

Unlike with `section-specs`, the `!` character does *not* have any special meaning (negation or otherwise) at the beginning of a `range-spec`!

Some example range specifications follow.

Match all `=for` html paragraphs:

```
/=for html/
```

Match all paragraphs between `=begin` html and `=end` html (note that this will *not* work correctly if such sections are nested):

```
/=begin html/..=/end html/
```

Match all paragraphs between the given `=item` name until the end of the current section:

```
/=item mine/..=/head\d/
```

Match all paragraphs between the given `=item` until the next item, or until the end of the itemized list (note that this will *not* work as desired if the item contains an itemized list nested within it):

```
/=item mine/..=(item|back)/
```

```
=end _NOT_IMPLEMENTED_
```

## OBJECT METHODS

The following methods are provided in this module. Each one takes a reference to the object itself as an implicit first parameter.

### `curr_headings()`

```
($head1, $head2, $head3, ...) = $parser->curr_headings();
$head1 = $parser->curr_headings(1);
```

This method returns a list of the currently active section headings and subheadings in the document being parsed. The list of headings returned corresponds to the most recently parsed paragraph of the input.

If an argument is given, it must correspond to the desired section heading number, in which case only the specified section heading is returned. If there is no current section heading at the specified level, then `undef` is returned.

### `select()`

```
$parser->select($section_spec1,$section_spec2,...);
```

This method is used to select the particular sections and subsections of POD documentation that are to be printed and/or processed. The existing set of selected sections is *replaced* with the given set of sections. See `add_selection()` for adding to the current set of selected sections.

Each of the `$section_spec` arguments should be a section specification as described in ["SECTION SPECIFICATIONS"](#). The section specifications are parsed by this method and the resulting regular expressions are stored in the invoking object.

If no `$section_spec` arguments are given, then the existing set of selected sections is cleared out (which means all sections will be processed).

This method should *not* normally be overridden by subclasses.

### `add_selection()`

```
$parser->add_selection($section_spec1,$section_spec2,...);
```

This method is used to add to the currently selected sections and subsections of POD documentation that are to be printed and/or processed. See `<select()` for replacing the currently selected sections.

Each of the `$section_spec` arguments should be a section specification as described in *"SECTION SPECIFICATIONS"*. The section specifications are parsed by this method and the resulting regular expressions are stored in the invoking object.

This method should *not* normally be overridden by subclasses.

```
clear_selections()
 $parser->clear_selections();
```

This method takes no arguments, it has the exact same effect as invoking `<select()` with no arguments.

```
match_section()
 $boolean = $parser->match_section($heading1,$heading2,...);
```

Returns a value of true if the given section and subsection heading titles match any of the currently selected section specifications in effect from prior calls to **select()** and **add\_selection()** (or if there are no explicitly selected/deselected sections).

The arguments `$heading1`, `$heading2`, etc. are the heading titles of the corresponding sections, subsections, etc. to try and match. If `$headingN` is omitted then it defaults to the current corresponding section heading title in the input.

This method should *not* normally be overridden by subclasses.

```
is_selected()
 $boolean = $parser->is_selected($paragraph);
```

This method is used to determine if the block of text given in `$paragraph` falls within the currently selected set of POD sections and subsections to be printed or processed. This method is also responsible for keeping track of the current input section and subsections. It is assumed that `$paragraph` is the most recently read (but not yet processed) input paragraph.

The value returned will be true if the `$paragraph` and the rest of the text in the same section as `$paragraph` should be selected (included) for processing; otherwise a false value is returned.

## EXPORTED FUNCTIONS

The following functions are exported by this module. Please note that these are functions (not methods) and therefore do *not* take an implicit first argument.

```
podselect()
 podselect(\%options,@filelist);
```

**podselect** will print the raw (untranslated) POD paragraphs of all POD sections in the given input files specified by `@filelist` according to the given options.

If any argument to **podselect** is a reference to a hash (associative array) then the values with the following keys are processed as follows:

### -output

A string corresponding to the desired output file (or `"&STDOUT"` or `"&STDERR"`) . The default is to use standard output.

### -sections

A reference to an array of sections specifications (as described in *"SECTION SPECIFICATIONS"*) which indicate the desired set of POD sections and subsections to be selected from input. If no section specifications are given, then all sections of the PODs are used.

=begin `_NOT_IMPLEMENTED_`

### -ranges

A reference to an array of range specifications (as described in *"RANGE SPECIFICATIONS"*) which indicate the desired range of POD paragraphs to be selected from the desired input sections. If no range

specifications are given, then all paragraphs of the desired sections are used.

```
=end _NOT_IMPLEMENTED_
```

All other arguments should correspond to the names of input files containing POD sections. A file name of "-" or "<STDIN" will be interpreted to mean standard input (which is the default if no filenames are given).

## PRIVATE METHODS AND DATA

**Pod::Select** makes use of a number of internal methods and data fields which clients should not need to see or use. For the sake of avoiding name collisions with client data and methods, these methods and fields are briefly discussed here. Determined hackers may obtain further information about them by reading the **Pod::Select** source code.

Private data fields are stored in the hash-object whose reference is returned by the **new()** constructor for this class. The names of all private methods and data-fields used by **Pod::Select** begin with a prefix of "\_" and match the regular expression `/^\_w+$/`.

```
_compile_section_spec()
```

```
$listref = $parser->_compile_section_spec($section_spec);
```

This function (note it is a function and *not* a method) takes a section specification (as described in *"SECTION SPECIFICATIONS"*) given in `$section_spec`, and compiles it into a list of regular expressions. If `$section_spec` has no syntax errors, then a reference to the list (array) of corresponding regular expressions is returned; otherwise `undef` is returned and an error message is printed (using **carp**) for each invalid regex.

```
=end _PRIVATE_
```

```
$self->{_SECTION_HEADINGS}
```

A reference to an array of the current section heading titles for each heading level (note that the first heading level title is at index 0).

```
=end _PRIVATE_
```

```
$self->{_SELECTED_SECTIONS}
```

A reference to an array of references to arrays. Each subarray is a list of anchored regular expressions (preceded by a "!" if the expression is to be negated). The index of the expression in the subarray should correspond to the index of the heading title in `$self->{_SECTION_HEADINGS}` that it is to be matched against.

```
=end _PRIVATE_
```

## SEE ALSO

*Pod::Parser*

## AUTHOR

Brad Appleton <bradapp@enteract.com>

Based on code for **pod2text** written by Tom Christiansen <tchrist@mox.perl.com>

**NAME**

Pod::Text – Convert POD data to formatted ASCII text

**SYNOPSIS**

```
use Pod::Text;
my $parser = Pod::Text->new (sentence => 0, width => 78);

Read POD from STDIN and write to STDOUT.
$parser->parse_from_filehandle;

Read POD from file.pod and write to file.txt.
$parser->parse_from_file ('file.pod', 'file.txt');
```

**DESCRIPTION**

Pod::Text is a module that can convert documentation in the POD format (the preferred language for documenting Perl) into formatted ASCII. It uses no special formatting controls or codes whatsoever, and its output is therefore suitable for nearly any device.

As a derived class from Pod::Parser, Pod::Text supports the same methods and interfaces. See [Pod::Parser](#) for all the details; briefly, one creates a new parser with `< Pod::Text->new()` and then calls either `parse_from_filehandle()` or `parse_from_file()`.

`new()` can take options, in the form of key/value pairs, that control the behavior of the parser. The currently recognized options are:

**alt** If set to a true value, selects an alternate output format that, among other things, uses a different heading style and marks `=item` entries with a colon in the left margin. Defaults to false.

**code**

If set to a true value, the non-POD parts of the input file will be included in the output. Useful for viewing code documented with POD blocks with the POD rendered and the code left intact.

**indent**

The number of spaces to indent regular text, and the default indentation for `=over` blocks. Defaults to 4.

**loose**

If set to a true value, a blank line is printed after a `=head1` heading. If set to false (the default), no blank line is printed after `=head1`, although one is still printed after `=head2`. This is the default because it's the expected formatting for manual pages; if you're formatting arbitrary text documents, setting this to true may result in more pleasing output.

**quotes**

Sets the quote marks used to surround `C< text`. If the value is a single character, it is used as both the left and right quote; if it is two characters, the first character is used as the left quote and the second as the right quote; and if it is four characters, the first two are used as the left quote and the second two as the right quote.

This may also be set to the special value `none`, in which case no quote marks are added around `C< text`.

**sentence**

If set to a true value, Pod::Text will assume that each sentence ends in two spaces, and will try to preserve that spacing. If set to false, all consecutive whitespace in non-verbatim paragraphs is compressed into a single space. Defaults to true.

**width**

The column at which to wrap text on the right-hand side. Defaults to 76.

The standard `Pod::Parser` method `parse_from_filehandle()` takes up to two arguments, the first being the file handle to read POD from and the second being the file handle to write the formatted output to. The first defaults to `STDIN` if not given, and the second defaults to `STDOUT`. The method `parse_from_file()` is almost identical, except that its two arguments are the input and output disk files instead. See [Pod::Parser](#) for the specific details.

## DIAGNOSTICS

Bizarre space in item

Item called without tag

(W) Something has gone wrong in internal `=item` processing. These messages indicate a bug in `Pod::Text`; you should never see them.

Can't open %s for reading: %s

(F) `Pod::Text` was invoked via the compatibility mode `pod2text()` interface and the input file it was given could not be opened.

Invalid quote specification "%s"

(F) The quote specification given (the `quotes` option to the constructor) was invalid. A quote specification must be one, two, or four characters long.

%s:%d: Unknown command paragraph: %s

(W) The POD source contained a non-standard command paragraph (something of the form `=command args`) that `Pod::Man` didn't know about. It was ignored.

%s:%d: Unknown escape: %s

(W) The POD source contained an `E<>` escape that `Pod::Text` didn't know about.

%s:%d: Unknown formatting code: %s

(W) The POD source contained a non-standard formatting code (something of the form `X<>`) that `Pod::Text` didn't know about.

%s:%d: Unmatched `=back`

(W) `Pod::Text` encountered a `=back` command that didn't correspond to an `=over` command.

## RESTRICTIONS

Embedded `Ctrl-As` (octal 001) in the input will be mapped to spaces on output, due to an internal implementation detail.

## NOTES

This is a replacement for an earlier `Pod::Text` module written by Tom Christiansen. It has a revamped interface, since it now uses `Pod::Parser`, but an interface roughly compatible with the old `Pod::Text::pod2text()` function is still available. Please change to the new calling convention, though.

The original `Pod::Text` contained code to do formatting via `termcap` sequences, although it wasn't turned on by default and it was problematic to get it to work at all. This rewrite doesn't even try to do that, but a subclass of it does. Look for [Pod::Text::Termcap](#).

## SEE ALSO

[Pod::Parser](#), [Pod::Text::Termcap](#), [pod2text\(1\)](#)

The current version of this module is always available from its web site at <http://www.eyrie.org/~eagle/software/podlators/>. It is also part of the Perl core distribution as of 5.6.0.

## AUTHOR

Russ Allbery <[rra@stanford.edu](mailto:rra@stanford.edu)>, based very heavily on the original `Pod::Text` by Tom Christiansen <[tchrist@mox.perl.com](mailto:tchrist@mox.perl.com)> and its conversion to `Pod::Parser` by Brad Appleton <[bradapp@enteract.com](mailto:bradapp@enteract.com)>.

**COPYRIGHT AND LICENSE**

Copyright 1999, 2000, 2001, 2002 by Russ Allbery <rra@stanford.edu>.

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

**NAME**

Pod::Text::Color – Convert POD data to formatted color ASCII text

**SYNOPSIS**

```
use Pod::Text::Color;
my $parser = Pod::Text::Color->new (sentence => 0, width => 78);

Read POD from STDIN and write to STDOUT.
$parser->parse_from_filehandle;

Read POD from file.pod and write to file.txt.
$parser->parse_from_file ('file.pod', 'file.txt');
```

**DESCRIPTION**

Pod::Text::Color is a simple subclass of Pod::Text that highlights output text using ANSI color escape sequences. Apart from the color, it in all ways functions like Pod::Text. See [Pod::Text](#) for details and available options.

Term::ANSIColor is used to get colors and therefore must be installed to use this module.

**BUGS**

This is just a basic proof of concept. It should be seriously expanded to support configurable coloration via options passed to the constructor, and **pod2text** should be taught about those.

**SEE ALSO**

[Pod::Text](#), [Pod::Parser](#)

The current version of this module is always available from its web site at <http://www.eyrie.org/~eagle/software/podlators/>. It is also part of the Perl core distribution as of 5.6.0.

**AUTHOR**

Russ Allbery <rra@stanford.edu>.

**COPYRIGHT AND LICENSE**

Copyright 1999, 2001 by Russ Allbery <rra@stanford.edu>.

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.



**NAME**

Pod::Text::Overstrike – Convert POD data to formatted overstrike text

**SYNOPSIS**

```
use Pod::Text::Overstrike;
my $parser = Pod::Text::Overstrike->new (sentence => 0, width => 78);

Read POD from STDIN and write to STDOUT.
$parser->parse_from_filehandle;

Read POD from file.pod and write to file.txt.
$parser->parse_from_file ('file.pod', 'file.txt');
```

**DESCRIPTION**

Pod::Text::Overstrike is a simple subclass of Pod::Text that highlights output text using overstrike sequences, in a manner similar to nroff. Characters in bold text are overstruck (character, backspace, character) and characters in underlined text are converted to overstruck underscores (underscore, backspace, character). This format was originally designed for hardcopy terminals and/or lineprinters, yet is readable on softcopy (CRT) terminals.

Overstruck text is best viewed by page-at-a-time programs that take advantage of the terminal's **stand-out** and *underline* capabilities, such as the less program on Unix.

Apart from the overstrike, it in all ways functions like Pod::Text. See [Pod::Text](#) for details and available options.

**BUGS**

Currently, the outermost formatting instruction wins, so for example underlined text inside a region of bold text is displayed as simply bold. There may be some better approach possible.

**SEE ALSO**

[Pod::Text](#), [Pod::Parser](#)

The current version of this module is always available from its web site at <http://www.eyrie.org/~eagle/software/podlators/>. It is also part of the Perl core distribution as of 5.6.0.

**AUTHOR**

Joe Smith <Joe.Smith@inwap.com>, using the framework created by Russ Allbery <rra@stanford.edu>.

**COPYRIGHT AND LICENSE**

Copyright 2000 by Joe Smith <Joe.Smith@inwap.com>. Copyright 2001 by Russ Allbery <rra@stanford.edu>.

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

**NAME**

Pod::Text::Termcap – Convert POD data to ASCII text with format escapes

**SYNOPSIS**

```
use Pod::Text::Termcap;
my $parser = Pod::Text::Termcap->new (sentence => 0, width => 78);

Read POD from STDIN and write to STDOUT.
$parser->parse_from_filehandle;

Read POD from file.pod and write to file.txt.
$parser->parse_from_file ('file.pod', 'file.txt');
```

**DESCRIPTION**

Pod::Text::Termcap is a simple subclass of Pod::Text that highlights output text using the correct termcap escape sequences for the current terminal. Apart from the format codes, it in all ways functions like Pod::Text. See *Pod::Text* for details and available options.

**NOTES**

This module uses Term::Cap to retrieve the formatting escape sequences for the current terminal, and falls back on the ECMA-48 (the same in this regard as ANSI X3.64 and ISO 6429, the escape codes also used by DEC VT100 terminals) if the bold, underline, and reset codes aren't set in the termcap information.

**SEE ALSO**

*Pod::Text*, *Pod::Parser*, *Term::Cap*

The current version of this module is always available from its web site at <http://www.eyrie.org/~eagle/software/podlators/>. It is also part of the Perl core distribution as of 5.6.0.

**AUTHOR**

Russ Allbery <rra@stanford.edu>.

**COPYRIGHT AND LICENSE**

Copyright 1999, 2001, 2002 by Russ Allbery <rra@stanford.edu>.

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

**NAME**

Pod::Usage, pod2usage() – print a usage message from embedded pod documentation

**SYNOPSIS**

```
use Pod::Usage

my $message_text = "This text precedes the usage message.";
my $exit_status = 2; ## The exit status to use
my $verbose_level = 0; ## The verbose level to use
my $filehandle = *STDERR; ## The filehandle to write to

pod2usage($message_text);

pod2usage($exit_status);

pod2usage({ -message => $message_text ,
 -exitval => $exit_status ,
 -verbose => $verbose_level,
 -output => $filehandle });

pod2usage(-msg => $message_text ,
 -exitval => $exit_status ,
 -verbose => $verbose_level,
 -output => $filehandle);
```

**ARGUMENTS**

**pod2usage** should be given either a single argument, or a list of arguments corresponding to an associative array (a "hash"). When a single argument is given, it should correspond to exactly one of the following:

- A string containing the text of a message to print *before* printing the usage message
- A numeric value corresponding to the desired exit status
- A reference to a hash

If more than one argument is given then the entire argument list is assumed to be a hash. If a hash is supplied (either as a reference or as a list) it should contain one or more elements with the following keys:

**-message**

**-msg**

The text of a message to print immediately prior to printing the program's usage message.

**-exitval**

The desired exit status to pass to the **exit()** function. This should be an integer, or else the string "NOEXIT" to indicate that control should simply be returned without terminating the invoking process.

**-verbose**

The desired level of "verboseness" to use when printing the usage message. If the corresponding value is 0, then only the "SYNOPSIS" section of the pod documentation is printed. If the corresponding value is 1, then the "SYNOPSIS" section, along with any section entitled "OPTIONS", "ARGUMENTS", or "OPTIONS AND ARGUMENTS" is printed. If the corresponding value is 2 or more then the entire manpage is printed.

**-output**

A reference to a filehandle, or the pathname of a file to which the usage message should be written. The default is `\*STDERR` unless the exit value is less than 2 (in which case the default is `\*STDOUT`).

**-input**

A reference to a filehandle, or the pathname of a file from which the invoking script's pod documentation should be read. It defaults to the file indicated by \$0 (\$PROGRAM\_NAME for users of *English.pm*).

**-pathlist**

A list of directory paths. If the input file does not exist, then it will be searched for in the given directory list (in the order the directories appear in the list). It defaults to the list of directories implied by \$ENV{PATH}. The list may be specified either by a reference to an array, or by a string of directory paths which use the same path separator as \$ENV{PATH} on your system (e.g., : for Unix, ; for MSWin32 and DOS).

**DESCRIPTION**

**pod2usage** will print a usage message for the invoking script (using its embedded pod documentation) and then exit the script with the desired exit status. The usage message printed may have any one of three levels of "verboseness": If the verbose level is 0, then only a synopsis is printed. If the verbose level is 1, then the synopsis is printed along with a description (if present) of the command line options and arguments. If the verbose level is 2, then the entire manual page is printed.

Unless they are explicitly specified, the default values for the exit status, verbose level, and output stream to use are determined as follows:

- If neither the exit status nor the verbose level is specified, then the default is to use an exit status of 2 with a verbose level of 0.
- If an exit status *is* specified but the verbose level is *not*, then the verbose level will default to 1 if the exit status is less than 2 and will default to 0 otherwise.
- If an exit status is *not* specified but verbose level *is* given, then the exit status will default to 2 if the verbose level is 0 and will default to 1 otherwise.
- If the exit status used is less than 2, then output is printed on STDOUT. Otherwise output is printed on STDERR.

Although the above may seem a bit confusing at first, it generally does "the right thing" in most situations. This determination of the default values to use is based upon the following typical Unix conventions:

- An exit status of 0 implies "success". For example, **diff(1)** exits with a status of 0 if the two files have the same contents.
- An exit status of 1 implies possibly abnormal, but non-defective, program termination. For example, **grep(1)** exits with a status of 1 if it did *not* find a matching line for the given regular expression.
- An exit status of 2 or more implies a fatal error. For example, **ls(1)** exits with a status of 2 if you specify an illegal (unknown) option on the command line.
- Usage messages issued as a result of bad command-line syntax should go to STDERR. However, usage messages issued due to an explicit request to print usage (like specifying **-help** on the command line) should go to STDOUT, just in case the user wants to pipe the output to a pager (such as **more(1)**).
- If program usage has been explicitly requested by the user, it is often desirable to exit with a status of 1 (as opposed to 0) after issuing the user-requested usage message. It is also desirable to give a more verbose description of program usage in this case.

**pod2usage** doesn't force the above conventions upon you, but it will use them by default if you don't expressly tell it to do otherwise. The ability of **pod2usage()** to accept a single number or a string makes it convenient to use as an innocent looking error message handling function:

```
use Pod::Usage;
use Getopt::Long;
```

```
Parse options
GetOptions("help", "man", "flag1") || pod2usage(2);
pod2usage(1) if ($opt_help);
pod2usage(-verbose => 2) if ($opt_man);

Check for too many filenames
pod2usage("$0: Too many files given.\n") if (@ARGV > 1);
```

Some user's however may feel that the above "economy of expression" is not particularly readable nor consistent and may instead choose to do something more like the following:

```
use Pod::Usage;
use Getopt::Long;

Parse options
GetOptions("help", "man", "flag1") || pod2usage(-verbose => 0);
pod2usage(-verbose => 1) if ($opt_help);
pod2usage(-verbose => 2) if ($opt_man);

Check for too many filenames
pod2usage(-verbose => 2, -message => "$0: Too many files given.\n")
 if (@ARGV > 1);
```

As with all things in Perl, *there's more than one way to do it*, and **pod2usage()** adheres to this philosophy. If you are interested in seeing a number of different ways to invoke **pod2usage** (although by no means exhaustive), please refer to *"EXAMPLES"*.

## EXAMPLES

Each of the following invocations of **pod2usage()** will print just the "SYNOPSIS" section to STDERR and will exit with a status of 2:

```
pod2usage();
pod2usage(2);
pod2usage(-verbose => 0);
pod2usage(-exitval => 2);
pod2usage({-exitval => 2, -output => *STDERR});
pod2usage({-verbose => 0, -output => *STDERR});
pod2usage(-exitval => 2, -verbose => 0);
pod2usage(-exitval => 2, -verbose => 0, -output => *STDERR);
```

Each of the following invocations of **pod2usage()** will print a message of "Syntax error." (followed by a newline) to STDERR, immediately followed by just the "SYNOPSIS" section (also printed to STDERR) and will exit with a status of 2:

```
pod2usage("Syntax error.");
pod2usage(-message => "Syntax error.", -verbose => 0);
pod2usage(-msg => "Syntax error.", -exitval => 2);
pod2usage({-msg => "Syntax error.", -exitval => 2, -output => *STDERR});
pod2usage({-msg => "Syntax error.", -verbose => 0, -output => *STDERR});
pod2usage(-msg => "Syntax error.", -exitval => 2, -verbose => 0);
pod2usage(-message => "Syntax error.",
 -exitval => 2,
 -verbose => 0,
```

```
-output => *STDERR);
```

Each of the following invocations of `pod2usage()` will print the "SYNOPSIS" section and any "OPTIONS" and/or "ARGUMENTS" sections to `STDOUT` and will exit with a status of 1:

```
pod2usage(1);
pod2usage(-verbose => 1);
pod2usage(-exitval => 1);
pod2usage({-exitval => 1, -output => *STDOUT});
pod2usage({-verbose => 1, -output => *STDOUT});
pod2usage(-exitval => 1, -verbose => 1);
pod2usage(-exitval => 1, -verbose => 1, -output => *STDOUT);
```

Each of the following invocations of `pod2usage()` will print the entire manual page to `STDOUT` and will exit with a status of 1:

```
pod2usage(-verbose => 2);
pod2usage({-verbose => 2, -output => *STDOUT});
pod2usage(-exitval => 1, -verbose => 2);
pod2usage({-exitval => 1, -verbose => 2, -output => *STDOUT});
```

### Recommended Use

Most scripts should print some type of usage message to `STDERR` when a command line syntax error is detected. They should also provide an option (usually `-H` or `-help`) to print a (possibly more verbose) usage message to `STDOUT`. Some scripts may even wish to go so far as to provide a means of printing their complete documentation to `STDOUT` (perhaps by allowing a `-man` option). The following complete example uses **Pod::Usage** in combination with **Getopt::Long** to do all of these things:

```
use Getopt::Long;
use Pod::Usage;

my $man = 0;
my $help = 0;
Parse options and print usage if there is a syntax error,
or if usage was explicitly requested.
GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
pod2usage(1) if $help;
pod2usage(-verbose => 2) if $man;

If no arguments were given, then allow STDIN to be used only
if it's not connected to a terminal (otherwise print usage)
pod2usage("$0: No files given.") if ((@ARGV == 0) && (-t STDIN));
__END__

=head1 NAME

sample - Using GetOpt::Long and Pod::Usage

=head1 SYNOPSIS

sample [options] [file ...]

Options:
 -help brief help message
 -man full documentation
```

```
=head1 OPTIONS

=over 8

=item B<-help>

Print a brief help message and exits.

=item B<-man>

Prints the manual page and exits.

=back

=head1 DESCRIPTION

B<This program> will read the given input file(s) and do something
useful with the contents thereof.

=cut
```

## CAVEATS

By default, **pod2usage()** will use `$0` as the path to the pod input file. Unfortunately, not all systems on which Perl runs will set `$0` properly (although if `$0` isn't found, **pod2usage()** will search `$ENV{PATH}` or else the list specified by the `-pathlist` option). If this is the case for your system, you may need to explicitly specify the path to the pod docs for the invoking script using something similar to the following:

```
pod2usage(-exitval => 2, -input => "/path/to/your/pod/docs");
```

## AUTHOR

Brad Appleton <bradapp@enteract.com>

Based on code for **Pod::Text::pod2text()** written by Tom Christiansen <tchrist@mox.perl.com>

## ACKNOWLEDGEMENTS

Steven McDougall <swmcd@world.std.com> for his help and patience with re-writing this manpage.

**NAME**

POSIX – Perl interface to IEEE Std 1003.1

**SYNOPSIS**

```
use POSIX;
use POSIX qw(setsid);
use POSIX qw(:errno_h :fcntl_h);

printf "EINTR is %d\n", EINTR;

$sess_id = POSIX::setsid();

$fd = POSIX::open($path, O_CREAT|O_EXCL|O_WRONLY, 0644);
note: that's a filedescriptor, *NOT* a filehandle
```

**DESCRIPTION**

The POSIX module permits you to access all (or nearly all) the standard POSIX 1003.1 identifiers. Many of these identifiers have been given Perl-ish interfaces. Things which are `#defines` in C, like `EINTR` or `O_NDELAY`, are automatically exported into your namespace. All functions are only exported if you ask for them explicitly. Most likely people will prefer to use the fully-qualified function names.

This document gives a condensed list of the features available in the POSIX module. Consult your operating system's manpages for general information on most features. Consult *perlfunc* for functions which are noted as being identical to Perl's builtin functions.

The first section describes POSIX functions from the 1003.1 specification. The second section describes some classes for signal objects, TTY objects, and other miscellaneous objects. The remaining sections list various constants and macros in an organization which roughly follows IEEE Std 1003.1b–1993.

**NOTE**

The POSIX module is probably the most complex Perl module supplied with the standard distribution. It incorporates autoloading, namespace games, and dynamic loading of code that's in Perl, C, or both. It's a great source of wisdom.

**CAVEATS**

A few functions are not implemented because they are C specific. If you attempt to call these, they will print a message telling you that they aren't implemented, and suggest using the Perl equivalent should one exist. For example, trying to access the `setjmp()` call will elicit the message "`setjmp()` is C-specific: use `eval {}` instead".

Furthermore, some evil vendors will claim 1003.1 compliance, but in fact are not so: they will not pass the PCTS (POSIX Compliance Test Suites). For example, one vendor may not define `EDEADLK`, or the semantics of the `errno` values set by `open(2)` might not be quite right. Perl does not attempt to verify POSIX compliance. That means you can currently successfully say "use POSIX", and then later in your program you find that your vendor has been lax and there's no usable `ICANON` macro after all. This could be construed to be a bug.

**FUNCTIONS**

|                    |                                                                                                                                                                                                                                              |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>_exit</code> | This is identical to the C function <code>_exit()</code> . It exits the program immediately which means among other things buffered I/O is <b>not</b> flushed.                                                                               |
| <code>abort</code> | This is identical to the C function <code>abort()</code> . It terminates the process with a <code>SIGABRT</code> signal unless caught by a signal handler or if the handler does not return normally (it e.g. does a <code>longjmp</code> ). |
| <code>abs</code>   | This is identical to Perl's builtin <code>abs()</code> function, returning the absolute value of its numerical argument.                                                                                                                     |



|         |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| access  | <p>Determines the accessibility of a file.</p> <pre> if( POSIX::access( "/", &amp;POSIX::R_OK ) ){     print "have read permission\n"; } </pre> <p>Returns undef on failure. Note: do not use <code>access()</code> for security purposes. Between the <code>access()</code> call and the operation you are preparing for the permissions might change: a classic <i>race condition</i>.</p>                                                                                                                                                                          |
| acos    | <p>This is identical to the C function <code>acos()</code>, returning the arcus cosine of its numerical argument. See also <a href="#">Math::Trig</a>.</p>                                                                                                                                                                                                                                                                                                                                                                                                            |
| alarm   | <p>This is identical to Perl's builtin <code>alarm()</code> function, either for arming or disarming the SIGALRM timer.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| asctime | <p>This is identical to the C function <code>asctime()</code>. It returns a string of the form</p> <pre> "Fri Jun  2 18:22:13 2000\n\0" </pre> <p>and it is called thusly</p> <pre> \$asctime = asctime(\$sec, \$min, \$hour, \$mday, \$mon, \$year,                   \$yday, \$yday, \$isdst); </pre> <p>The <code>\$mon</code> is zero-based: January equals 0. The <code>\$year</code> is 1900-based: 2001 equals 101. The <code>\$yday</code>, <code>\$yday</code>, and <code>\$isdst</code> default to zero (and the first two are usually ignored anyway).</p> |
| asin    | <p>This is identical to the C function <code>asin()</code>, returning the arcus sine of its numerical argument. See also <a href="#">Math::Trig</a>.</p>                                                                                                                                                                                                                                                                                                                                                                                                              |
| assert  | <p>Unimplemented, but you can use <a href="#">die</a> and the <a href="#">Carp</a> module to achieve similar things.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| atan    | <p>This is identical to the C function <code>atan()</code>, returning the arcus tangent of its numerical argument. See also <a href="#">Math::Trig</a>.</p>                                                                                                                                                                                                                                                                                                                                                                                                           |
| atan2   | <p>This is identical to Perl's builtin <code>atan2()</code> function, returning the arcus tangent defined by its two numerical arguments, the <i>y</i> coordinate and the <i>x</i> coordinate. See also <a href="#">Math::Trig</a>.</p>                                                                                                                                                                                                                                                                                                                               |
| atexit  | <p><code>atexit()</code> is C-specific: use <code>END {}</code> instead, see <a href="#">perlsub</a>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| atof    | <p><code>atof()</code> is C-specific. Perl converts strings to numbers transparently. If you need to force a scalar to a number, add a zero to it.</p>                                                                                                                                                                                                                                                                                                                                                                                                                |
| atoi    | <p><code>atoi()</code> is C-specific. Perl converts strings to numbers transparently. If you need to force a scalar to a number, add a zero to it. If you need to have just the integer part, see <a href="#">int</a>.</p>                                                                                                                                                                                                                                                                                                                                            |
| atol    | <p><code>atol()</code> is C-specific. Perl converts strings to numbers transparently. If you need to force a scalar to a number, add a zero to it. If you need to have just the integer part, see <a href="#">int</a>.</p>                                                                                                                                                                                                                                                                                                                                            |
| bsearch | <p><code>bsearch()</code> not supplied. For doing binary search on wordlists, see <a href="#">Search::Dict</a>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| calloc  | <p><code>calloc()</code> is C-specific. Perl does memory management transparently.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| ceil    | <p>This is identical to the C function <code>ceil()</code>, returning the smallest integer value greater than or equal to the given numerical argument.</p>                                                                                                                                                                                                                                                                                                                                                                                                           |
| chdir   | <p>This is identical to Perl's builtin <code>chdir()</code> function, allowing one to change the working (default) directory, see <a href="#">chdir</a>.</p>                                                                                                                                                                                                                                                                                                                                                                                                          |
| chmod   | <p>This is identical to Perl's builtin <code>chmod()</code> function, allowing one to change file and directory permissions, see <a href="#">chmod</a>.</p>                                                                                                                                                                                                                                                                                                                                                                                                           |

- chown** This is identical to Perl's builtin `chown()` function, allowing one to change file and directory owners and groups, see [chown](#).
- clearerr** Use the method `IO::Handle::clearerr()` instead, to reset the error state (if any) and EOF state (if any) of the given stream.
- clock** This is identical to the C function `clock()`, returning the amount of spent processor time in microseconds.
- close** Close the file. This uses file descriptors such as those obtained by calling `POSIX::open`.
- ```
$fd = POSIX::open( "foo", &POSIX::O_RDONLY );
POSIX::close( $fd );
```
- Returns `undef` on failure.
- See also [close](#).
- closedir** This is identical to Perl's builtin `closedir()` function for closing a directory handle, see [closedir](#).
- cos** This is identical to Perl's builtin `cos()` function, for returning the cosine of its numerical argument, see [cos](#). See also [Math::Trig](#).
- cosh** This is identical to the C function `cosh()`, for returning the hyperbolic cosine of its numeric argument. See also [Math::Trig](#).
- creat** Create a new file. This returns a file descriptor like the ones returned by `POSIX::open`. Use `POSIX::close` to close the file.
- ```
$fd = POSIX::creat("foo", 0611);
POSIX::close($fd);
```
- See also [sysopen](#) and its `O_CREAT` flag.
- ctermid** Generates the path name for the controlling terminal.
- ```
$path = POSIX::ctermid();
```
- ctime** This is identical to the C function `ctime()` and equivalent to `asctime(localtime(...))`, see [asctime](#) and [localtime](#).
- cuserid** Get the login name of the owner of the current process.
- ```
$name = POSIX::cuserid();
```
- difftime** This is identical to the C function `difftime()`, for returning the time difference (in seconds) between two times (as returned by `time()`), see [time](#).
- div** `div()` is C-specific, use [int](#) on the usual `/` division and the modulus `%`.
- dup** This is similar to the C function `dup()`, for duplicating a file descriptor.
- This uses file descriptors such as those obtained by calling `POSIX::open`.
- Returns `undef` on failure.
- dup2** This is similar to the C function `dup2()`, for duplicating a file descriptor to an another known file descriptor.
- This uses file descriptors such as those obtained by calling `POSIX::open`.
- Returns `undef` on failure.
- errno** Returns the value of `errno`.
- ```
$errno = POSIX::errno();
```

	This identical to the numerical values of the <code>\$!</code> , see \$ERRNO .
<code>execl</code>	<code>execl()</code> is C-specific, see exec .
<code>execle</code>	<code>execle()</code> is C-specific, see exec .
<code>execlp</code>	<code>execlp()</code> is C-specific, see exec .
<code>execv</code>	<code>execv()</code> is C-specific, see exec .
<code>execve</code>	<code>execve()</code> is C-specific, see exec .
<code>execvp</code>	<code>execvp()</code> is C-specific, see exec .
<code>exit</code>	This is identical to Perl's builtin <code>exit()</code> function for exiting the program, see exit .
<code>exp</code>	This is identical to Perl's builtin <code>exp()</code> function for returning the exponent (<i>e</i> -based) of the numerical argument, see exp .
<code>fabs</code>	This is identical to Perl's builtin <code>abs()</code> function for returning the absolute value of the numerical argument, see abs .
<code>fclose</code>	Use method <code>IO::Handle::close()</code> instead, or see close .
<code>fcntl</code>	This is identical to Perl's builtin <code>fcntl()</code> function, see fcntl .
<code>fdopen</code>	Use method <code>IO::Handle::new_from_fd()</code> instead, or see open .
<code>feof</code>	Use method <code>IO::Handle::eof()</code> instead, or see eof .
<code>ferror</code>	Use method <code>IO::Handle::error()</code> instead.
<code>fflush</code>	Use method <code>IO::Handle::flush()</code> instead. See also \$OUTPUT_AUTOFLUSH .
<code>fgetc</code>	Use method <code>IO::Handle::getc()</code> instead, or see read .
<code>fgetpos</code>	Use method <code>IO::Seekable::getpos()</code> instead, or see seek .
<code>fgets</code>	Use method <code>IO::Handle::gets()</code> instead. Similar to <code><></code> , also known as readline .
<code>fileno</code>	Use method <code>IO::Handle::fileno()</code> instead, or see fileno .
<code>floor</code>	This is identical to the C function <code>floor()</code> , returning the largest integer value less than or equal to the numerical argument.
<code>fmod</code>	This is identical to the C function <code>fmod()</code> . <pre>\$r = fmod(\$x, \$y);</pre> <p>It returns the remainder $\\$r = \\$x - \\$n * \\y, where $\\$n = \text{trunc}(\\$x / \\$y)$. The $\\$r$ has the same sign as $\\$x$ and magnitude (absolute value) less than the magnitude of $\\$y$.</p>
<code>fdopen</code>	Use method <code>IO::File::open()</code> instead, or see open .
<code>fork</code>	This is identical to Perl's builtin <code>fork()</code> function for duplicating the current process, see fork and perlfork if you are in Windows.
<code>fpathconf</code>	Retrieves the value of a configurable limit on a file or directory. This uses file descriptors such as those obtained by calling <code>POSIX::open</code> . <p>The following will determine the maximum length of the longest allowable pathname on the filesystem which holds <code>/tmp/foo</code>.</p> <pre>\$fd = POSIX::open("/tmp/foo", &POSIX::O_RDONLY); \$path_max = POSIX::fpathconf(\$fd, &POSIX::_PC_PATH_MAX);</pre> <p>Returns <code>undef</code> on failure.</p>

<code>fprintf</code>	<code>fprintf()</code> is C-specific, see printf instead.
<code>fputc</code>	<code>fputc()</code> is C-specific, see print instead.
<code>fputs</code>	<code>fputs()</code> is C-specific, see print instead.
<code>fread</code>	<code>fread()</code> is C-specific, see read instead.
<code>free</code>	<code>free()</code> is C-specific. Perl does memory management transparently.
<code>freopen</code>	<code>freopen()</code> is C-specific, see open instead.
<code>frexp</code>	Return the mantissa and exponent of a floating-point number. <pre>(\$mantissa, \$exponent) = POSIX::frexp(1.234e56);</pre>
<code>fscanf</code>	<code>fscanf()</code> is C-specific, use <code><></code> and regular expressions instead.
<code>fseek</code>	Use method <code>IO::Seekable::seek()</code> instead, or see seek .
<code>fsetpos</code>	Use method <code>IO::Seekable::setpos()</code> instead, or see seek .
<code>fstat</code>	Get file status. This uses file descriptors such as those obtained by calling <code>POSIX::open</code> . The data returned is identical to the data from Perl's builtin <code>stat</code> function. <pre>\$fd = POSIX::open("foo", &POSIX::O_RDONLY); @stats = POSIX::fstat(\$fd);</pre>
<code>ftell</code>	Use method <code>IO::Seekable::tell()</code> instead, or see tell .
<code>fwrite</code>	<code>fwrite()</code> is C-specific, see print instead.
<code>getc</code>	This is identical to Perl's builtin <code>getc()</code> function, see getc .
<code>getchar</code>	Returns one character from STDIN. Identical to Perl's <code>getc()</code> , see getc .
<code>getcwd</code>	Returns the name of the current working directory. See also Cwd .
<code>getegid</code>	Returns the effective group identifier. Similar to Perl's builtin variable <code>\$<</code> , see \$EGID .
<code>getenv</code>	Returns the value of the specified environment variable. The same information is available through the <code>%ENV</code> array.
<code>geteuid</code>	Returns the effective user identifier. Identical to Perl's builtin <code>\$></code> variable, see \$EUID .
<code>getgid</code>	Returns the user's real group identifier. Similar to Perl's builtin variable <code>\$)</code> , see \$GID .
<code>getgrgid</code>	This is identical to Perl's builtin <code>getgrgid()</code> function for returning group entries by group identifiers, see getgrgid .
<code>getgrnam</code>	This is identical to Perl's builtin <code>getgrnam()</code> function for returning group entries by group names, see getgrnam .
<code>getgroups</code>	Returns the ids of the user's supplementary groups. Similar to Perl's builtin variable <code>\$)</code> , see \$GID .
<code>getlogin</code>	This is identical to Perl's builtin <code>getlogin()</code> function for returning the user name associated with the current session, see getlogin .
<code>getpgrp</code>	This is identical to Perl's builtin <code>getpgrp()</code> function for returning the process group identifier of the current process, see getpgrp .
<code>getpid</code>	Returns the process identifier. Identical to Perl's builtin variable <code>\$\$</code> , see \$PID .
<code>getppid</code>	This is identical to Perl's builtin <code>getppid()</code> function for returning the process identifier of the parent process of the current process, see getppid .

getpwnam

This is identical to Perl's builtin `getpwnam()` function for returning user entries by user names, see [getpwnam](#).

getpwuid

This is identical to Perl's builtin `getpwuid()` function for returning user entries by user identifiers, see [getpwuid](#).

gets

Returns one line from STDIN, similar to `<>`, also known as the `readline()` function, see [readline](#).

NOTE: if you have C programs that still use `gets()`, be very afraid. The `gets()` function is a source of endless grief because it has no buffer overrun checks. It should **never** be used. The `fgets()` function should be preferred instead.

getuid

Returns the user's identifier. Identical to Perl's builtin `$<` variable, see [\\$UID](#).

gmtime

This is identical to Perl's builtin `gmtime()` function for converting seconds since the epoch to a date in Greenwich Mean Time, see [gmtime](#).

isalnum

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:alnum:]]/` construct instead, or possibly the `/\w/` construct.

isalpha

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:alpha:]]/` construct instead.

isatty

Returns a boolean indicating whether the specified filehandle is connected to a tty. Similar to the `-t` operator, see [-X](#).

isctrl

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:cntrl:]]/` construct instead.

isdigit

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:digit:]]/` construct instead, or the `/\d/` construct.

isgraph

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:graph:]]/` construct instead.

islower

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:lower:]]/` construct instead. Do **not** use `/[a-z]/`.

isprint

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:print:]]/` construct instead.

ispunct

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:punct:]]/` construct instead.

isspace

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:space:]]/` construct instead, or the `/\s/` construct. (Note that `/\s/` and `/[[:space:]]/` are slightly different in that `/[[:space:]]/` can normally match a vertical tab, while `/\s/` does not.)

isupper

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:upper:]]/` construct instead. Do **not** use `/[A-Z]/`.

isxdigit

This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using regular expressions and the `/[[:xdigit:]]/` construct instead, or simply `/[0-9a-f]/i`.

kill This is identical to Perl's builtin `kill()` function for sending signals to processes (often to terminate them), see [kill](#).

labs (For returning absolute values of long integers.) `labs()` is C-specific, see [abs](#) instead.

ldexp This is identical to the C function `ldexp()` for multiplying floating point numbers with powers of two.

```
$x_quadrupled = POSIX::ldexp($x, 2);
```

ldiv (For computing dividends of long integers.) `ldiv()` is C-specific, use `/` and `int()` instead.

link This is identical to Perl's builtin `link()` function for creating hard links into files, see [link](#).

localeconv

Get numeric formatting information. Returns a reference to a hash containing the current locale formatting values.

Here is how to query the database for the **de** (Deutsch or German) locale.

```
$loc = POSIX::setlocale( &POSIX::LC_ALL, "de" );
print "Locale = $loc\n";
$lconv = POSIX::localeconv();
print "decimal_point    = ", $lconv->{decimal_point},    "\n";
print "thousands_sep    = ", $lconv->{thousands_sep},    "\n";
print "grouping          = ", $lconv->{grouping},          "\n";
print "int_curr_symbol    = ", $lconv->{int_curr_symbol},  "\n";
print "currency_symbol    = ", $lconv->{currency_symbol},  "\n";
print "mon_decimal_point = ", $lconv->{mon_decimal_point}, "\n";
print "mon_thousands_sep = ", $lconv->{mon_thousands_sep}, "\n";
print "mon_grouping       = ", $lconv->{mon_grouping},     "\n";
print "positive_sign     = ", $lconv->{positive_sign},    "\n";
print "negative_sign     = ", $lconv->{negative_sign},    "\n";
print "int_frac_digits    = ", $lconv->{int_frac_digits},  "\n";
print "frac_digits       = ", $lconv->{frac_digits},      "\n";
print "p_cs_precedes      = ", $lconv->{p_cs_precedes},   "\n";
print "p_sep_by_space     = ", $lconv->{p_sep_by_space},   "\n";
print "n_cs_precedes      = ", $lconv->{n_cs_precedes},   "\n";
print "n_sep_by_space     = ", $lconv->{n_sep_by_space},   "\n";
print "p_sign_posn        = ", $lconv->{p_sign_posn},     "\n";
print "n_sign_posn        = ", $lconv->{n_sign_posn},     "\n";
```

localtime This is identical to Perl's builtin `localtime()` function for converting seconds since the epoch to a date see [localtime](#).

log This is identical to Perl's builtin `log()` function, returning the natural (*e*-based) logarithm of the numerical argument, see [log](#).

log10 This is identical to the C function `log10()`, returning the 10-base logarithm of the numerical argument. You can also use

```
sub log10 { log($_[0]) / log(10) }
```

or

```
sub log10 { log($_[0]) / 2.30258509299405 }
```

or

```
sub log10 { log($_[0]) * 0.434294481903252 }
```

- longjmp** `longjmp()` is C-specific: use [die](#) instead.
- lseek** Move the file's read/write position. This uses file descriptors such as those obtained by calling `POSIX::open`.
- ```
$fd = POSIX::open("foo", &POSIX::O_RDONLY);
$off_t = POSIX::lseek($fd, 0, &POSIX::SEEK_SET);
```
- Returns `undef` on failure.
- malloc** `malloc()` is C-specific. Perl does memory management transparently.
- mblen** This is identical to the C function `mblen()`. Perl does not have any support for the wide and multibyte characters of the C standards, so this might be a rather useless function.
- mbstowcs** This is identical to the C function `mbstowcs()`. Perl does not have any support for the wide and multibyte characters of the C standards, so this might be a rather useless function.
- mbtowc** This is identical to the C function `mbtowc()`. Perl does not have any support for the wide and multibyte characters of the C standards, so this might be a rather useless function.
- memchr** `memchr()` is C-specific, see [index](#) instead.
- memcmp** `memcmp()` is C-specific, use `eq` instead, see [perlop](#).
- memcpy** `memcpy()` is C-specific, use `=`, see [perlop](#), or see [substr](#).
- memmove** `memmove()` is C-specific, use `=`, see [perlop](#), or see [substr](#).
- memset** `memset()` is C-specific, use `x` instead, see [perlop](#).
- mkdir** This is identical to Perl's builtin `mkdir()` function for creating directories, see [mkdir](#).
- mkfifo** This is similar to the C function `mkfifo()` for creating FIFO special files.
- ```
if (mkfifo($path, $mode)) { ....
```
- Returns `undef` on failure. The `$mode` is similar to the mode of `mkdir()`, see [mkdir](#).
- mktime** Convert date/time info to a calendar time.
- Synopsis:
- ```
mktime(sec, min, hour, mday, mon, year, wday = 0, yday = 0, isdst = 0)
```
- The month (`mon`), weekday (`wday`), and yearday (`yday`) begin at zero. I.e. January is 0, not 1; Sunday is 0, not 1; January 1st is 0, not 1. The year (`year`) is given in years since 1900. I.e. The year 1995 is 95; the year 2001 is 101. Consult your system's `mktime()` manpage for details about these and the other arguments.
- Calendar time for December 12, 1995, at 10:30 am.
- ```
$time_t = POSIX::mktime( 0, 30, 10, 12, 11, 95 );
print "Date = ", POSIX::ctime($time_t);
```
- Returns `undef` on failure.
- modf** Return the integral and fractional parts of a floating-point number.
- ```
($fractional, $integral) = POSIX::modf(3.14);
```
- nice** This is similar to the C function `nice()`, for changing the scheduling preference of the current process. Positive arguments mean more polite process, negative values more needy process. Normal user processes can only be more polite.

- Returns `undef` on failure.
- offsetof** `offsetof()` is C-specific, you probably want to see [pack](#) instead.
- open** Open a file for reading or writing. This returns file descriptors, not Perl filehandles. Use `POSIX::close` to close the file.
- Open a file read-only with mode 0666.
- ```
$fd = POSIX::open( "foo" );
```
- Open a file for read and write.
- ```
$fd = POSIX::open("foo", &POSIX::O_RDWR);
```
- Open a file for write, with truncation.
- ```
$fd = POSIX::open( "foo", &POSIX::O_WRONLY | &POSIX::O_TRUNC );
```
- Create a new file with mode 0640. Set up the file for writing.
- ```
$fd = POSIX::open("foo", &POSIX::O_CREAT | &POSIX::O_WRONLY, 0640);
```
- Returns `undef` on failure.
- See also [sysopen](#).
- opendir** Open a directory for reading.
- ```
$dir = POSIX::opendir( "/tmp" );
@files = POSIX::readdir( $dir );
POSIX::closedir( $dir );
```
- Returns `undef` on failure.
- pathconf** Retrieves the value of a configurable limit on a file or directory.
- The following will determine the maximum length of the longest allowable pathname on the filesystem which holds `/tmp`.
- ```
$path_max = POSIX::pathconf("/tmp", &POSIX::_PC_PATH_MAX);
```
- Returns `undef` on failure.
- pause** This is similar to the C function `pause()`, which suspends the execution of the current process until a signal is received.
- Returns `undef` on failure.
- perror** This is identical to the C function `perror()`, which outputs to the standard error stream the specified message followed by `": "` and the current error string. Use the `warn()` function and the `$!` variable instead, see [warn](#) and [\\$ERRNO](#).
- pipe** Create an interprocess channel. This returns file descriptors like those returned by `POSIX::open`.
- ```
($fd0, $fd1) = POSIX::pipe();
POSIX::write( $fd0, "hello", 5 );
POSIX::read( $fd1, $buf, 5 );
```
- See also [pipe](#).
- pow** Computes `$x` raised to the power `$exponent`.
- ```
$ret = POSIX::pow($x, $exponent);
```
- You can also use the `**` operator, see [perlop](#).



- printf**      Formats and prints the specified arguments to STDOUT. See also [printf](#).
- putc**        `putc()` is C-specific, see [print](#) instead.
- putchar**    `putchar()` is C-specific, see [print](#) instead.
- puts**        `puts()` is C-specific, see [print](#) instead.
- qsort**        `qsort()` is C-specific, see [sort](#) instead.
- raise**        Sends the specified signal to the current process. See also [kill](#) and the `$$` in [\\$PID](#).
- rand**        `rand()` is non-portable, see [rand](#) instead.
- read**        Read from a file. This uses file descriptors such as those obtained by calling `POSIX::open`. If the buffer `$buf` is not large enough for the read then Perl will extend it to make room for the request.
- ```

        $fd = POSIX::open( "foo", &POSIX::O_RDONLY );
        $bytes = POSIX::read( $fd, $buf, 3 );

```
- Returns `undef` on failure.
- See also [sysread](#).
- readdir** This is identical to Perl's builtin `readdir()` function for reading directory entries, see [readdir](#).
- realloc** `realloc()` is C-specific. Perl does memory management transparently.
- remove** This is identical to Perl's builtin `unlink()` function for removing files, see [unlink](#).
- rename** This is identical to Perl's builtin `rename()` function for renaming files, see [rename](#).
- rewind** Seeks to the beginning of the file.
- rewinddir** This is identical to Perl's builtin `rewinddir()` function for rewinding directory entry streams, see [rewinddir](#).
- rmdir** This is identical to Perl's builtin `rmdir()` function for removing (empty) directories, see [rmdir](#).
- scanf** `scanf()` is C-specific, use `<>` and regular expressions instead, see [perlre](#).
- setgid** Sets the real group identifier and the effective group identifier for this process. Similar to assigning a value to the Perl's builtin `$)` variable, see [\\$GID](#), except that the latter will change only the real user identifier, and that the `setgid()` uses only a single numeric argument, as opposed to a space-separated list of numbers.
- setjmp** `setjmp()` is C-specific: use `eval {}` instead, see [eval](#).
- setlocale** Modifies and queries program's locale. The following examples assume
- ```

 use POSIX qw(setlocale LC_ALL LC_CTYPE);

```
- has been issued.
- The following will set the traditional UNIX system locale behavior (the second argument "C").
- ```

        $loc = setlocale( LC_ALL, "C" );

```
- The following will query the current `LC_CTYPE` category. (No second argument means 'query'.)
- ```

 $loc = setlocale(LC_CTYPE);

```
- The following will set the `LC_CTYPE` behaviour according to the locale environment variables (the second argument ""). Please see your systems `setlocale(3)` documentation for the

locale environment variables' meaning or consult [perllocale](#).

```
$loc = setlocale(LC_CTYPE, "");
```

The following will set the LC\_COLLATE behaviour to Argentinian Spanish. **NOTE:** The naming and availability of locales depends on your operating system. Please consult [perllocale](#) for how to find out which locales are available in your system.

```
$loc = setlocale(LC_ALL, "es_AR.ISO8859-1");
```

**setpgid** This is similar to the C function `setpgid()` for setting the process group identifier of the current process.

Returns undef on failure.

**setsid** This is identical to the C function `setsid()` for setting the session identifier of the current process.

**setuid** Sets the real user identifier and the effective user identifier for this process. Similar to assigning a value to the Perl's builtin `$<` variable, see [\\$UID](#), except that the latter will change only the real user identifier.

**sigaction** Detailed signal management. This uses `POSIX::SigAction` objects for the `action` and `oldaction` arguments. Consult your system's `sigaction` manpage for details.

Synopsis:

```
sigaction(sig, action, oldaction = 0)
```

Returns undef on failure.

**siglongjmp**

`siglongjmp()` is C-specific: use [die](#) instead.

**sigpending**

Examine signals that are blocked and pending. This uses `POSIX::SigSet` objects for the `sigset` argument. Consult your system's `sigpending` manpage for details.

Synopsis:

```
sigpending(sigset)
```

Returns undef on failure.

**sigprocmask**

Change and/or examine calling process's signal mask. This uses `POSIX::SigSet` objects for the `sigset` and `oldsigset` arguments. Consult your system's `sigprocmask` manpage for details.

Synopsis:

```
sigprocmask(how, sigset, oldsigset = 0)
```

Returns undef on failure.

**sigsetjmp** `sigsetjmp()` is C-specific: use `eval {}` instead, see [eval](#).

**sigsuspend**

Install a signal mask and suspend process until signal arrives. This uses `POSIX::SigSet` objects for the `signal_mask` argument. Consult your system's `sigsuspend` manpage for details.

Synopsis:

```
sigsuspend(signal_mask)
```

|                       |                                                                                                                                                                                                                                                                                                                                                                                                 |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                       | Returns undef on failure.                                                                                                                                                                                                                                                                                                                                                                       |
| <code>sin</code>      | This is identical to Perl's builtin <code>sin()</code> function for returning the sine of the numerical argument, see <a href="#">sin</a> . See also <a href="#">Math::Trig</a> .                                                                                                                                                                                                               |
| <code>sinh</code>     | This is identical to the C function <code>sinh()</code> for returning the hyperbolic sine of the numerical argument. See also <a href="#">Math::Trig</a> .                                                                                                                                                                                                                                      |
| <code>sleep</code>    | This is functionally identical to Perl's builtin <code>sleep()</code> function for suspending the execution of the current process for certain number of seconds, see <a href="#">sleep</a> . There is one significant difference, however: <code>POSIX::sleep()</code> returns the number of <b>unslept</b> seconds, while the <code>CORE::sleep()</code> returns the number of slept seconds. |
| <code>sprintf</code>  | This is similar to Perl's builtin <code>sprintf()</code> function for returning a string that has the arguments formatted as requested, see <a href="#">sprintf</a> .                                                                                                                                                                                                                           |
| <code>sqrt</code>     | This is identical to Perl's builtin <code>sqrt()</code> function. for returning the square root of the numerical argument, see <a href="#">sqrt</a> .                                                                                                                                                                                                                                           |
| <code>srand</code>    | Give a seed the pseudorandom number generator, see <a href="#">srand</a> .                                                                                                                                                                                                                                                                                                                      |
| <code>sscanf</code>   | <code>sscanf()</code> is C-specific, use regular expressions instead, see <a href="#">perlre</a> .                                                                                                                                                                                                                                                                                              |
| <code>stat</code>     | This is identical to Perl's builtin <code>stat()</code> function for returning information about files and directories.                                                                                                                                                                                                                                                                         |
| <code>strcat</code>   | <code>strcat()</code> is C-specific, use <code>.</code> instead, see <a href="#">perlop</a> .                                                                                                                                                                                                                                                                                                   |
| <code>strchr</code>   | <code>strchr()</code> is C-specific, see <a href="#">index</a> instead.                                                                                                                                                                                                                                                                                                                         |
| <code>strcmp</code>   | <code>strcmp()</code> is C-specific, use <code>eq</code> or <code>cmp</code> instead, see <a href="#">perlop</a> .                                                                                                                                                                                                                                                                              |
| <code>strcoll</code>  | This is identical to the C function <code>strcoll()</code> for collating (comparing) strings transformed using the <code>strxfrm()</code> function. Not really needed since Perl can do this transparently, see <a href="#">perllocale</a> .                                                                                                                                                    |
| <code>strcpy</code>   | <code>strcpy()</code> is C-specific, use <code>=</code> instead, see <a href="#">perlop</a> .                                                                                                                                                                                                                                                                                                   |
| <code>strcspn</code>  | <code>strcspn()</code> is C-specific, use regular expressions instead, see <a href="#">perlre</a> .                                                                                                                                                                                                                                                                                             |
| <code>strerror</code> | Returns the error string for the specified <code>errno</code> . Identical to the string form of the <code>\$!</code> , see <a href="#">\$ERRNO</a> .                                                                                                                                                                                                                                            |
| <code>strftime</code> | Convert date and time information to string. Returns the string.                                                                                                                                                                                                                                                                                                                                |

Synopsis:

```
strftime(fmt, sec, min, hour, mday, mon, year, wday = -1, yday = -1,
```

The month (`mon`), weekday (`wday`), and yearday (`yday`) begin at zero. I.e. January is 0, not 1; Sunday is 0, not 1; January 1st is 0, not 1. The year (`year`) is given in years since 1900. I.e., the year 1995 is 95; the year 2001 is 101. Consult your system's `strftime()` manpage for details about these and the other arguments. If you want your code to be portable, your format (`fmt`) argument should use only the conversion specifiers defined by the ANSI C standard. These are `aAbBcdHIjmMpSUWxXyYZ%`. The given arguments are made consistent as though by calling `mktime()` before calling your system's `strftime()` function, except that the `isdst` value is not affected.

The string for Tuesday, December 12, 1995.

```
$str = POSIX::strftime("%A, %B %d, %Y", 0, 0, 0, 12, 11, 95, 2);
print "$str\n";
```

- strlen**     `strlen()` is C-specific, use `length()` instead, see [length](#).
- strncat**     `strncat()` is C-specific, use `.` instead, see [perlop](#).
- strncmp**     `strncmp()` is C-specific, use `eq` instead, see [perlop](#).
- strncpy**     `strncpy()` is C-specific, use `=` instead, see [perlop](#).
- strpbrk**     `strpbrk()` is C-specific, use regular expressions instead, see [perlre](#).
- strrchr**     `strrchr()` is C-specific, see [rindex](#) instead.
- strspn**     `strspn()` is C-specific, use regular expressions instead, see [perlre](#).
- strstr**     This is identical to Perl's builtin `index()` function, see [index](#).
- strtod**     String to double translation. Returns the parsed number and the number of characters in the unparsed portion of the string. Truly POSIX-compliant systems set `$!` (`$ERRNO`) to indicate a translation error, so clear `$!` before calling `strtod`. However, non-POSIX systems may not check for overflow, and therefore will never set `$!`.
- `strtod` should respect any POSIX `setlocale()` settings.
- To parse a string `$str` as a floating point number use
- ```
$! = 0;
($num, $n_unparsed) = POSIX::strtod($str);
```
- The second returned item and `$!` can be used to check for valid input:
- ```
if (($str eq '') || ($n_unparsed != 0) || !$!) {
 die "Non-numeric input $str" . $! ? ": $!\n" : "\n";
}
```
- When called in a scalar context `strtod` returns the parsed number.
- strtok**     `strtok()` is C-specific, use regular expressions instead, see [perlre](#), or [split](#).
- strtol**     String to (long) integer translation. Returns the parsed number and the number of characters in the unparsed portion of the string. Truly POSIX-compliant systems set `$!` (`$ERRNO`) to indicate a translation error, so clear `$!` before calling `strtol`. However, non-POSIX systems may not check for overflow, and therefore will never set `$!`.
- `strtol` should respect any POSIX `setlocale()` settings.
- To parse a string `$str` as a number in some base `$base` use
- ```
$! = 0;
($num, $n_unparsed) = POSIX::strtol($str, $base);
```
- The base should be zero or between 2 and 36, inclusive. When the base is zero or omitted `strtol` will use the string itself to determine the base: a leading "0x" or "0X" means hexadecimal; a leading "0" means octal; any other leading characters mean decimal. Thus, "1234" is parsed as a decimal number, "01234" as an octal number, and "0x1234" as a hexadecimal number.
- The second returned item and `$!` can be used to check for valid input:
- ```
if (($str eq '') || ($n_unparsed != 0) || !$!) {
 die "Non-numeric input $str" . $! ? ": $!\n" : "\n";
}
```
- When called in a scalar context `strtol` returns the parsed number.
- strtoul**     String to unsigned (long) integer translation. `strtoul()` is identical to `strtol()` except that `strtoul()` only parses unsigned integers. See [strtol](#) for details.

Note: Some vendors supply `strtod()` and `strtol()` but not `strtoul()`. Other vendors that do supply `strtoul()` parse "-1" as a valid value.

**strxfrm** String transformation. Returns the transformed string.

```
$dst = POSIX::strxfrm($src);
```

Used in conjunction with the `strcoll()` function, see [/strcoll](#).

Not really needed since Perl can do this transparently, see [perllocale](#).

**sysconf** Retrieves values of system configurable variables.

The following will get the machine's clock speed.

```
$clock_ticks = POSIX::sysconf(&POSIX::_SC_CLK_TCK);
```

Returns `undef` on failure.

**system** This is identical to Perl's builtin `system()` function, see [system](#).

**tan** This is identical to the C function `tan()`, returning the tangent of the numerical argument. See also [Math::Trig](#).

**tanh** This is identical to the C function `tanh()`, returning the hyperbolic tangent of the numerical argument. See also [Math::Trig](#).

**tcdrain** This is similar to the C function `tcdrain()` for draining the output queue of its argument stream.

Returns `undef` on failure.

**tcflow** This is similar to the C function `tcflow()` for controlling the flow of its argument stream.

Returns `undef` on failure.

**tcflush** This is similar to the C function `tcflush()` for flushing the I/O buffers of its argument stream.

Returns `undef` on failure.

**tcgetpgrp** This is identical to the C function `tcgetpgrp()` for returning the process group identifier of the foreground process group of the controlling terminal.

**tcsendbreak**

This is similar to the C function `tcsendbreak()` for sending a break on its argument stream.

Returns `undef` on failure.

**tcsetpgrp** This is similar to the C function `tcsetpgrp()` for setting the process group identifier of the foreground process group of the controlling terminal.

Returns `undef` on failure.

**time** This is identical to Perl's builtin `time()` function for returning the number of seconds since the epoch (whatever it is for the system), see [time](#).

**times** The `times()` function returns elapsed realtime since some point in the past (such as system startup), user and system times for this process, and user and system times used by child processes. All times are returned in clock ticks.

```
($realtime, $user, $system, $cuser, $csystem) = POSIX::times();
```

Note: Perl's builtin `times()` function returns four values, measured in seconds.

**tmpfile** Use method `IO::File::new_tmpfile()` instead, or see [File::Temp](#).

**tmpnam** Returns a name for a temporary file.

```
$tmpfile = POSIX::tmpnam();
```

For security reasons, which are probably detailed in your system's documentation for the C library `tmpnam()` function, this interface should not be used; instead see [File::Temp](#).

**tolower** This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using the `lc()` function, see [lc](#), or the equivalent `\L` operator inside doublequotish strings.

**toupper** This is identical to the C function, except that it can apply to a single character or to a whole string. Consider using the `uc()` function, see [uc](#), or the equivalent `\U` operator inside doublequotish strings.

**ttyname** This is identical to the C function `ttyname()` for returning the name of the current terminal.

**tzname** Retrieves the time conversion information from the `tzname` variable.

```
POSIX::tzset();
($std, $dst) = POSIX::tzname();
```

**tzset** This is identical to the C function `tzset()` for setting the current timezone based on the environment variable `TZ`, to be used by `ctime()`, `localtime()`, `mktime()`, and `strftime()` functions.

**umask** This is identical to Perl's builtin `umask()` function for setting (and querying) the file creation permission mask, see [umask](#).

**uname** Get name of current operating system.

```
($sysname, $nodename, $release, $version, $machine) = POSIX::uname();
```

Note that the actual meanings of the various fields are not that well standardized, do not expect any great portability. The `$sysname` might be the name of the operating system, the `$nodename` might be the name of the host, the `$release` might be the (major) release number of the operating system, the `$version` might be the (minor) release number of the operating system, and the `$machine` might be a hardware identifier. Maybe.

**ungetc** Use method `IO::Handle::ungetc()` instead.

**unlink** This is identical to Perl's builtin `unlink()` function for removing files, see [unlink](#).

**utime** This is identical to Perl's builtin `utime()` function for changing the time stamps of files and directories, see [utime](#).

**vfprintf** `vfprintf()` is C-specific, see [printf](#) instead.

**vprintf** `vprintf()` is C-specific, see [printf](#) instead.

**vsprintf** `vsprintf()` is C-specific, see [sprintf](#) instead.

**wait** This is identical to Perl's builtin `wait()` function, see [wait](#).

**waitpid** Wait for a child process to change state. This is identical to Perl's builtin `waitpid()` function, see [waitpid](#).

```
$pid = POSIX::waitpid(-1, POSIX::WNOHANG);
print "status = ", ($? / 256), "\n";
```

**wcstombs**

This is identical to the C function `wcstombs()`. Perl does not have any support for the wide and multibyte characters of the C standards, so this might be a rather useless function.

**wctomb** This is identical to the C function `wctomb()`. Perl does not have any support for the wide and multibyte characters of the C standards, so this might be a rather useless function.

**write** Write to a file. This uses file descriptors such as those obtained by calling `POSIX::open`.

```
$fd = POSIX::open("foo", &POSIX::O_WRONLY);
$buf = "hello";
$bytes = POSIX::write($b, $buf, 5);
```

Returns `undef` on failure.

See also [syswrite](#).

## CLASSES

### POSIX::SigAction

**new** Creates a new `POSIX::SigAction` object which corresponds to the C struct `sigaction`. This object will be destroyed automatically when it is no longer needed. The first parameter is the fully-qualified name of a sub which is a signal-handler. The second parameter is a `POSIX::SigSet` object, it defaults to the empty set. The third parameter contains the `sa_flags`, it defaults to 0.

```
$sigset = POSIX::SigSet->new(SIGINT, SIGQUIT);
$sigaction = POSIX::SigAction->new('main::handler', $sigset, &POSIX::
```

This `POSIX::SigAction` object should be used with the `POSIX::sigaction()` function.

### POSIX::SigSet

**new** Create a new `SigSet` object. This object will be destroyed automatically when it is no longer needed. Arguments may be supplied to initialize the set.

Create an empty set.

```
$sigset = POSIX::SigSet->new;
```

Create a set with `SIGUSR1`.

```
$sigset = POSIX::SigSet->new(&POSIX::SIGUSR1);
```

**addset** Add a signal to a `SigSet` object.

```
$sigset->addset(&POSIX::SIGUSR2);
```

Returns `undef` on failure.

**delset** Remove a signal from the `SigSet` object.

```
$sigset->delset(&POSIX::SIGUSR2);
```

Returns `undef` on failure.

**emptyset** Initialize the `SigSet` object to be empty.

```
$sigset->emptyset();
```

Returns `undef` on failure.

**fillset** Initialize the `SigSet` object to include all signals.

```
$sigset->fillset();
```

Returns `undef` on failure.

**ismember**

Tests the SigSet object to see if it contains a specific signal.

```
if($sigset->ismember(&POSIX::SIGUSR1)){
 print "contains SIGUSR1\n";
}
```

**POSIX::Termios****new**

Create a new Termios object. This object will be destroyed automatically when it is no longer needed. A Termios object corresponds to the termios C struct. `new()` mallocs a new one, `getattr()` fills it from a file descriptor, and `setattr()` sets a file descriptor's parameters to match Termios' contents.

```
$termios = POSIX::Termios->new;
```

**getattr**

Get terminal control attributes.

Obtain the attributes for stdin.

```
$termios->getattr()
```

Obtain the attributes for stdout.

```
$termios->getattr(1)
```

Returns undef on failure.

**getcc**

Retrieve a value from the `c_cc` field of a termios object. The `c_cc` field is an array so an index must be specified.

```
$c_cc[1] = $termios->getcc(1);
```

**getcflag**

Retrieve the `c_cflag` field of a termios object.

```
$c_cflag = $termios->getcflag;
```

**getiflag**

Retrieve the `c_iflag` field of a termios object.

```
$c_iflag = $termios->getiflag;
```

**getispeed**

Retrieve the input baud rate.

```
$ispeed = $termios->getispeed;
```

**getlflag**

Retrieve the `c_lflag` field of a termios object.

```
$c_lflag = $termios->getlflag;
```

**getoflag**

Retrieve the `c_oflag` field of a termios object.

```
$c_oflag = $termios->getoflag;
```

**getospeed**

Retrieve the output baud rate.

```
$ospeed = $termios->getospeed;
```

**setattr**

Set terminal control attributes.

Set attributes immediately for stdout.

```
$termios->setattr(1, &POSIX::TCSANOW);
```

Returns undef on failure.



**setcc** Set a value in the `c_cc` field of a `termios` object. The `c_cc` field is an array so an index must be specified.

```
$termios->setcc(&POSIX::VEOF, 1);
```

**setcflag** Set the `c_cflag` field of a `termios` object.

```
$termios->setcflag($c_cflag | &POSIX::CLOCAL);
```

**setiflag** Set the `c_iflag` field of a `termios` object.

```
$termios->setiflag($c_iflag | &POSIX::BRKINT);
```

**setispeed** Set the input baud rate.

```
$termios->setispeed(&POSIX::B9600);
```

Returns `undef` on failure.

**setlflag** Set the `c_lflag` field of a `termios` object.

```
$termios->setlflag($c_lflag | &POSIX::ECHO);
```

**setoflag** Set the `c_oflag` field of a `termios` object.

```
$termios->setoflag($c_oflag | &POSIX::OPOST);
```

**setospeed**

Set the output baud rate.

```
$termios->setospeed(&POSIX::B9600);
```

Returns `undef` on failure.

**Baud rate values**

B38400 B75 B200 B134 B300 B1800 B150 B0 B19200 B1200 B9600 B600 B4800 B50 B2400  
B110

**Terminal interface values**

TCSADRAIN TCSANOW TCOON TCIOFLUSH TCOFLUSH TCION TCIFLUSH  
TCSAFLUSH TCIOFF TCOOFF

**c\_cc field values**

VEOF VEOL VERASE VINTR VKILL VQUIT VSUSP VSTART VSTOP VMIN VTIME  
NCCS

**c\_cflag field values**

CLOCAL CREAD CSIZE CS5 CS6 CS7 CS8 CSTOPB HUPCL PARENB PARODD

**c\_iflag field values**

BRKINT ICRNL IGNBRK IGNCR IGNPAR INLCR INPCK ISTRIP IXOFF IXON PARMRK

**c\_lflag field values**

ECHO ECHOE ECHOK ECHONL ICANON IEXTEN ISIG NOFLSH TOSTOP

**c\_oflag field values**

OPOST

## **PATHNAME CONSTANTS**

Constants

\_PC\_CHOWN\_RESTRICTED \_PC\_LINK\_MAX \_PC\_MAX\_CANON \_PC\_MAX\_INPUT  
\_PC\_NAME\_MAX \_PC\_NO\_TRUNC \_PC\_PATH\_MAX \_PC\_PIPE\_BUF \_PC\_VDISABLE

## POSIX CONSTANTS

### Constants

\_POSIX\_ARG\_MAX \_POSIX\_CHILD\_MAX \_POSIX\_CHOWN\_RESTRICTED  
\_POSIX\_JOB\_CONTROL \_POSIX\_LINK\_MAX \_POSIX\_MAX\_CANON  
\_POSIX\_MAX\_INPUT \_POSIX\_NAME\_MAX \_POSIX\_NGROUPS\_MAX  
\_POSIX\_NO\_TRUNC \_POSIX\_OPEN\_MAX \_POSIX\_PATH\_MAX \_POSIX\_PIPE\_BUF  
\_POSIX\_SAVED\_IDS \_POSIX\_SSIZE\_MAX \_POSIX\_STREAM\_MAX  
\_POSIX\_TZNAME\_MAX \_POSIX\_VDISABLE \_POSIX\_VERSION

## SYSTEM CONFIGURATION

### Constants

\_SC\_ARG\_MAX \_SC\_CHILD\_MAX \_SC\_CLK\_TCK \_SC\_JOB\_CONTROL  
\_SC\_NGROUPS\_MAX \_SC\_OPEN\_MAX \_SC\_PAGESIZE \_SC\_SAVED\_IDS  
\_SC\_STREAM\_MAX \_SC\_TZNAME\_MAX \_SC\_VERSION

## ERRNO

### Constants

E2BIG EACCES EADDRINUSE EADDRNOTAVAIL EAFNOSUPPORT EAGAIN  
EALREADY EBADF EBUSY ECHILD ECONNABORTED ECONNREFUSED  
ECONNRESET EDEADLK EDESTADDRREQ EDOM EDQUOT EEXIST EFAULT EFBIG  
EHOSTDOWN EHOSTUNREACH EINPROGRESS EINTR EINVAL EIO EISCONN EISDIR  
ELOOP EMFILE EMLINK EMSGSIZE ENAMETOOLONG ENETDOWN ENETRESET  
ENETUNREACH ENFILE ENOBUFS ENODEV ENOENT ENOEXEC ENOLCK ENOMEM  
ENOPROTOOPT ENOSPC ENOSYS ENOTBLK ENOTCONN ENOTDIR ENOTEMPTY  
ENOTSOCK ENOTTY ENXIO EOPNOTSUPP EPERM EPFNOSUPPORT EPIPE  
EPROCLIM EPROTONOSUPPORT EPROTOTYPE ERANGE EREMOTE ERESTART  
EROFS ESHUTDOWN ESOCKTNOSUPPORT ESPIPE ESRCH ESTALE ETIMEDOUT  
ETOOMANYREFS ETXTBSY EUSERS EWOULDBLOCK EXDEV

## FCNTL

### Constants

FD\_CLOEXEC F\_DUPFD F\_GETFD F\_GETFL F\_GETLK F\_OK F\_RDLCK F\_SETFD  
F\_SETFL F\_SETLK F\_SETLKW F\_UNLCK F\_WRLCK O\_ACCMODE O\_APPEND  
O\_CREAT O\_EXCL O\_NOCTTY O\_NONBLOCK O\_RDONLY O\_RDWR O\_TRUNC  
O\_WRONLY

## FLOAT

### Constants

DBL\_DIG DBL\_EPSILON DBL\_MANT\_DIG DBL\_MAX DBL\_MAX\_10\_EXP  
DBL\_MAX\_EXP DBL\_MIN DBL\_MIN\_10\_EXP DBL\_MIN\_EXP FLT\_DIG FLT\_EPSILON  
FLT\_MANT\_DIG FLT\_MAX FLT\_MAX\_10\_EXP FLT\_MAX\_EXP FLT\_MIN  
FLT\_MIN\_10\_EXP FLT\_MIN\_EXP FLT\_RADIX FLT\_ROUNDS LDBL\_DIG  
LDBL\_EPSILON LDBL\_MANT\_DIG LDBL\_MAX LDBL\_MAX\_10\_EXP  
LDBL\_MAX\_EXP LDBL\_MIN LDBL\_MIN\_10\_EXP LDBL\_MIN\_EXP

## LIMITS

### Constants

ARG\_MAX CHAR\_BIT CHAR\_MAX CHAR\_MIN CHILD\_MAX INT\_MAX INT\_MIN  
LINK\_MAX LONG\_MAX LONG\_MIN MAX\_CANON MAX\_INPUT MB\_LEN\_MAX  
NAME\_MAX NGROUPS\_MAX OPEN\_MAX PATH\_MAX PIPE\_BUF SCHAR\_MAX  
SCHAR\_MIN SHRT\_MAX SHRT\_MIN SSIZE\_MAX STREAM\_MAX TZNAME\_MAX  
UCHAR\_MAX UINT\_MAX ULONG\_MAX USHRT\_MAX

**LOCALE**

## Constants

LC\_ALL LC\_COLLATE LC\_CTYPE LC\_MONETARY LC\_NUMERIC LC\_TIME

**MATH**

## Constants

HUGE\_VAL

**SIGNAL**

## Constants

SA\_NOCLDSTOP SA\_NOCLDWAIT SA\_NODEFER SA\_ONSTACK SA\_RESETHAND  
SA\_RESTART SA\_SIGINFO SIGABRT SIGALRM SIGCHLD SIGCONT SIGFPE SIGHUP  
SIGILL SIGINT SIGKILL SIGPIPE SIGQUIT SIGSEGV SIGSTOP SIGTERM SIGTSTP  
SIGTTIN SIGTTOU SIGUSR1 SIGUSR2 SIG\_BLOCK SIG\_DFL SIG\_ERR SIG\_IGN  
SIG\_SETMASK SIG\_UNBLOCK**STAT**

## Constants

S\_IRGRP S\_IROTH S\_IRUSR S\_IRWXG S\_IRWXO S\_IRWXU S\_ISGID S\_ISUID  
S\_IWGRP S\_IWOTH S\_IWUSR S\_IXGRP S\_IXOTH S\_IXUSR

Macros S\_ISBLK S\_ISCHR S\_ISDIR S\_ISFIFO S\_ISREG

**STDLIB**

## Constants

EXIT\_FAILURE EXIT\_SUCCESS MB\_CUR\_MAX RAND\_MAX

**STDIO**

## Constants

BUFSIZ EOF FILENAME\_MAX L\_ctermid L\_cuserid L\_tmpname TMP\_MAX

**TIME**

## Constants

CLK\_TCK CLOCKS\_PER\_SEC

**UNISTD**

## Constants

R\_OK SEEK\_CUR SEEK\_END SEEK\_SET STDIN\_FILENO STDOUT\_FILENO  
STDERR\_FILENO W\_OK X\_OK**WAIT**

## Constants

WNOHANG WUNTRACED

WNOHANG Do not suspend the calling process until a child process changes state but instead return immediately.

WUNTRACED Catch stopped child processes.

Macros WIFEXITED WEXITSTATUS WIFSIGNALED WTERMSIG WIFSTOPPED WSTOPSIG

WIFEXITED WIFEXITED(\$?) returns true if the child process exited normally (exit()) or by falling off the end of main()

|             |                                                                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| WEXITSTATUS | WEXITSTATUS(\$?) returns the normal exit status of the child process (only meaningful if WIFEXITED(\$?) is true)                                        |
| WIFSIGNALED | WIFSIGNALED(\$?) returns true if the child process terminated because of a signal                                                                       |
| WTERMSIG    | WTERMSIG(\$?) returns the signal the child process terminated for (only meaningful if WIFSIGNALED(\$?) is true)                                         |
| WIFSTOPPED  | WIFSTOPPED(\$?) returns true if the child process is currently stopped (can happen only if you specified the WUNTRACED flag to <code>waitpid()</code> ) |
| WSTOPSIG    | WSTOPSIG(\$?) returns the signal the child process was stopped for (only meaningful if WIFSTOPPED(\$?) is true)                                         |

**NAME**

re – Perl pragma to alter regular expression behaviour

**SYNOPSIS**

```
use re 'taint';
($x) = ($^X =~ /^(.*)$/s); # $x is tainted here

$pat = '(?{ $foo = 1 })';
use re 'eval';
/foo${pat}bar/; # won't fail (when not under -T switch)

{
 no re 'taint'; # the default
 ($x) = ($^X =~ /^(.*)$/s); # $x is not tainted here

 no re 'eval'; # the default
 /foo${pat}bar/; # disallowed (with or without -T switch)
}

use re 'debug'; # NOT lexically scoped (as others are)
/^(.*)$/s; # output debugging info during
 # compile and run time

use re 'debugcolor'; # same as 'debug', but with colored output
...
```

(We use `$_X` in these examples because it's tainted by default.)

**DESCRIPTION**

When `use re 'taint'` is in effect, and a tainted string is the target of a regex, the regex memories (or values returned by the `m//` operator in list context) are tainted. This feature is useful when regex operations on tainted data aren't meant to extract safe substrings, but to perform other transformations.

When `use re 'eval'` is in effect, a regex is allowed to contain `(?{ ... })` zero-width assertions even if regular expression contains variable interpolation. That is normally disallowed, since it is a potential security risk. Note that this pragma is ignored when the regular expression is obtained from tainted data, i.e. evaluation is always disallowed with tainted regular expressions. See [\(?{ code }\)](#).

For the purpose of this pragma, interpolation of precompiled regular expressions (i.e., the result of `qr//`) is *not* considered variable interpolation. Thus:

```
/foo${pat}bar/
```

is allowed if `$pat` is a precompiled regular expression, even if `$pat` contains `(?{ ... })` assertions.

When `use re 'debug'` is in effect, perl emits debugging messages when compiling and using regular expressions. The output is the same as that obtained by running a `-DDEBUGGING`-enabled perl interpreter with the `-Dr` switch. It may be quite voluminous depending on the complexity of the match. Using `debugcolor` instead of `debug` enables a form of output that can be used to get a colorful display on terminals that understand termcap color sequences. Set `$ENV{PERL_RE_TC}` to a comma-separated list of termcap properties to use for highlighting strings on/off, pre-point part on/off. See [Debugging regular expressions in perldebug](#) for additional info.

The directive `use re 'debug'` is *not lexically scoped*, as the other directives are. It has both compile-time and run-time effects.

See [Pragmatic Modules](#).

**NAME**

Search::Dict, look – search for key in dictionary file

**SYNOPSIS**

```
use Search::Dict;
look *FILEHANDLE, $key, $dict, $fold;

use Search::Dict;
look *FILEHANDLE, $params;
```

**DESCRIPTION**

Sets file position in FILEHANDLE to be first line greater than or equal (stringwise) to *\$key*. Returns the new file position, or *-1* if an error occurs.

The flags specify dictionary order and case folding:

If *\$dict* is true, search by dictionary order (ignore anything but word characters and whitespace). The default is honour all characters.

If *\$fold* is true, ignore case. The default is to honour case.

If there are only three arguments and the third argument is a hash reference, the keys of that hash can have values *dict*, *fold*, and *comp* or *xfrm* (see below), and their corresponding values will be used as the parameters.

If a comparison subroutine (*comp*) is defined, it must return less than zero, zero, or greater than zero, if the first comparand is less than, equal, or greater than the second comparand.

If a transformation subroutine (*xfrm*) is defined, its value is used to transform the lines read from the filehandle before their comparison.

**NAME**

SelectSaver – save and restore selected file handle

**SYNOPSIS**

```
use SelectSaver;

{
 my $saver = new SelectSaver(FILEHANDLE);
 # FILEHANDLE is selected
}
previous handle is selected

{
 my $saver = new SelectSaver;
 # new handle may be selected, or not
}
previous handle is selected
```

**DESCRIPTION**

A `SelectSaver` object contains a reference to the file handle that was selected when it was created. If its `new` method gets an extra parameter, then that parameter is selected; otherwise, the selected file handle remains unchanged.

When a `SelectSaver` is destroyed, it re-selects the file handle that was selected when it was created.

## NAME

SelfLoader – load functions only on demand

## SYNOPSIS

```
package FOOBAR;
use SelfLoader;

... (initializing code)

__DATA__
sub {....
```

## DESCRIPTION

This module tells its users that functions in the FOOBAR package are to be autoloaded from after the `__DATA__` token. See also [Autoloading in perlsub](#).

### The `__DATA__` token

The `__DATA__` token tells the perl compiler that the perl code for compilation is finished. Everything after the `__DATA__` token is available for reading via the filehandle `FOOBAR::DATA`, where FOOBAR is the name of the current package when the `__DATA__` token is reached. This works just the same as `__END__` does in package 'main', but for other modules data after `__END__` is not automatically retrievable, whereas data after `__DATA__` is. The `__DATA__` token is not recognized in versions of perl prior to 5.001m.

Note that it is possible to have `__DATA__` tokens in the same package in multiple files, and that the last `__DATA__` token in a given package that is encountered by the compiler is the one accessible by the filehandle. This also applies to `__END__` and main, i.e. if the 'main' program has an `__END__`, but a module 'require'd (not 'use'd) by that program has a 'package main;' declaration followed by an `__DATA__`, then the `DATA` filehandle is set to access the data after the `__DATA__` in the module, not the data after the `__END__` token in the 'main' program, since the compiler encounters the 'require'd file later.

### SelfLoader autoloading

The **SelfLoader** works by the user placing the `__DATA__` token *after* perl code which needs to be compiled and run at 'require' time, but *before* subroutine declarations that can be loaded in later – usually because they may never be called.

The **SelfLoader** will read from the `FOOBAR::DATA` filehandle to load in the data after `__DATA__`, and load in any subroutine when it is called. The costs are the one-time parsing of the data after `__DATA__`, and a load delay for the `_first_` call of any autoloaded function. The benefits (hopefully) are a speeded up compilation phase, with no need to load functions which are never used.

The **SelfLoader** will stop reading from `__DATA__` if it encounters the `__END__` token – just as you would expect. If the `__END__` token is present, and is followed by the token `DATA`, then the **SelfLoader** leaves the `FOOBAR::DATA` filehandle open on the line after that token.

The **SelfLoader** exports the `AUTOLOAD` subroutine to the package using the **SelfLoader**, and this loads the called subroutine when it is first called.

There is no advantage to putting subroutines which will `_always_` be called after the `__DATA__` token.

### Autoloading and package lexicals

A 'my `$pack_lexical`' statement makes the variable `$pack_lexical` local `_only_` to the file up to the `__DATA__` token. Subroutines declared elsewhere `_cannot_` see these types of variables, just as if you declared subroutines in the package but in another file, they cannot see these variables.

So specifically, autoloaded functions cannot see package lexicals (this applies to both the **SelfLoader** and the Autoloader). The `vars` pragma provides an alternative to defining package-level globals that will be visible to autoloaded routines. See the documentation on **vars** in the pragma section of [perlmod](#).



## SelfLoader and AutoLoader

The **SelfLoader** can replace the AutoLoader – just change ‘use AutoLoader’ to ‘use SelfLoader’ (though note that the **SelfLoader** exports the AUTOLOAD function – but if you have your own AUTOLOAD and are using the AutoLoader too, you probably know what you’re doing), and the `__END__` token to `__DATA__`. You will need perl version 5.001m or later to use this (version 5.001 with all patches up to patch m).

There is no need to inherit from the **SelfLoader**.

The **SelfLoader** works similarly to the AutoLoader, but picks up the subs from after the `__DATA__` instead of in the ‘lib/auto’ directory. There is a maintenance gain in not needing to run AutoSplit on the module at installation, and a runtime gain in not needing to keep opening and closing files to load subs. There is a runtime loss in needing to parse the code after the `__DATA__`. Details of the **AutoLoader** and another view of these distinctions can be found in that module’s documentation.

### `__DATA__`, `__END__`, and the `FOOBAR::DATA` filehandle.

This section is only relevant if you want to use the `FOOBAR::DATA` together with the **SelfLoader**.

Data after the `__DATA__` token in a module is read using the `FOOBAR::DATA` filehandle. `__END__` can still be used to denote the end of the `__DATA__` section if followed by the token `DATA` – this is supported by the **SelfLoader**. The `FOOBAR::DATA` filehandle is left open if an `__END__` followed by a `DATA` is found, with the filehandle positioned at the start of the line after the `__END__` token. If no `__END__` token is present, or an `__END__` token with no `DATA` token on the same line, then the filehandle is closed.

The **SelfLoader** reads from wherever the current position of the `FOOBAR::DATA` filehandle is, until the EOF or `__END__`. This means that if you want to use that filehandle (and ONLY if you want to), you should either

1. Put all your subroutine declarations immediately after the `__DATA__` token and put your own data after those declarations, using the `__END__` token to mark the end of subroutine declarations. You must also ensure that the **SelfLoader** reads first by calling `'SelfLoader->load_stubs()'`, or by using a function which is selfloaded;

or

2. You should read the `FOOBAR::DATA` filehandle first, leaving the handle open and positioned at the first line of subroutine declarations.

You could conceivably do both.

## Classes and inherited methods.

For modules which are not classes, this section is not relevant. This section is only relevant if you have methods which could be inherited.

A subroutine stub (or forward declaration) looks like

```
sub stub;
```

i.e. it is a subroutine declaration without the body of the subroutine. For modules which are not classes, there is no real need for stubs as far as autoloading is concerned.

For modules which ARE classes, and need to handle inherited methods, stubs are needed to ensure that the method inheritance mechanism works properly. You can load the stubs into the module at ‘require’ time, by adding the statement `'SelfLoader->load_stubs()'` to the module to do this.

The alternative is to put the stubs in before the `__DATA__` token BEFORE releasing the module, and for this purpose the `Devel::SelfStubber` module is available. However this does require the extra step of ensuring that the stubs are in the module. If this is done I strongly recommend that this is done BEFORE releasing the module – it should NOT be done at install time in general.

**Multiple packages and fully qualified subroutine names**

Subroutines in multiple packages within the same file are supported – but you should note that this requires exporting the `SelfLoader::AUTOLOAD` to every package which requires it. This is done automatically by the **SelfLoader** when it first loads the subs into the cache, but you should really specify it in the initialization before the `__DATA__` by putting a ‘use SelfLoader’ statement in each package.

Fully qualified subroutine names are also supported. For example,

```
__DATA__
sub foo::bar {23}
package baz;
sub dob {32}
```

will all be loaded correctly by the **SelfLoader**, and the **SelfLoader** will ensure that the packages ‘foo’ and ‘baz’ correctly have the **SelfLoader** `AUTOLOAD` method when the data after `__DATA__` is first parsed.

**NAME**

Shell – run shell commands transparently within perl

**SYNOPSIS**

See below.

**DESCRIPTION**

```
Date: Thu, 22 Sep 94 16:18:16 -0700
Message-Id: <9409222318.AA17072@scalpel.netlabs.com>
To: perl5-porters@isu.edu
From: Larry Wall <lwall@scalpel.netlabs.com>
Subject: a new module I just wrote
```

Here's one that'll whack your mind a little out.

```
#!/usr/bin/perl

use Shell;

$foo = echo("howdy", "<funny>", "world");
print $foo;

$passwd = cat("</etc/passwd");
print $passwd;

sub ps;
print ps -ww;

cp("/etc/passwd", "/tmp/passwd");
```

That's maybe too gonzo. It actually exports an AUTOLOAD to the current package (and uncovered a bug in Beta 3, by the way). Maybe the usual usage should be

```
use Shell qw(echo cat ps cp);
```

Larry

If you set `$Shell::capture_stderr` to 1, the module will attempt to capture the STDERR of the process as well.

The module now should work on Win32.

Jenda

There seemed to be a problem where all arguments to a shell command were quoted before being executed. As in the following example:

```
cat('</etc/passwd');
ls('*.*pl');
```

really turned into:

```
cat '</etc/passwd'
ls '*.*pl'
```

instead of:

```
cat </etc/passwd
ls *.*pl
```

and of course, this is wrong.

I have fixed this bug, it was brought up by Wolfgang Laun [ID 20000326.008]

Casey

**OBJECT ORIENTED SYNTAX**

Shell now has an OO interface. Good for namespace conservation and shell representation.

```
use Shell;
my $sh = Shell->new;
print $sh->ls;
```

Casey

**AUTHOR**

Larry Wall

Changes by Jenda@Krynicky.cz and Dave Cottle <d.cottle@csc.canterbury.ac.nz

Changes and bug fixes by Casey West <casey@geeknest.com

## NAME

sigtrap – Perl pragma to enable simple signal handling

## SYNOPSIS

```
use sigtrap;
use sigtrap qw(stack-trace old-interface-signals); # equivalent
use sigtrap qw(BUS SEGV PIPE ABRT);
use sigtrap qw(die INT QUIT);
use sigtrap qw(die normal-signals);
use sigtrap qw(die untrapped normal-signals);
use sigtrap qw(die untrapped normal-signals
 stack-trace any error-signals);
use sigtrap 'handler' => \&my_handler, 'normal-signals';
use sigtrap qw(handler my_handler normal-signals
 stack-trace error-signals);
```

## DESCRIPTION

The **sigtrap** pragma is a simple interface to installing signal handlers. You can have it install one of two handlers supplied by **sigtrap** itself (one which provides a Perl stack trace and one which simply `die()`s), or alternately you can supply your own handler for it to install. It can be told only to install a handler for signals which are either untrapped or ignored. It has a couple of lists of signals to trap, plus you can supply your own list of signals.

The arguments passed to the `use` statement which invokes **sigtrap** are processed in order. When a signal name or the name of one of **sigtrap**'s signal lists is encountered a handler is immediately installed, when an option is encountered it affects subsequently installed handlers.

## OPTIONS

### SIGNAL HANDLERS

These options affect which handler will be used for subsequently installed signals.

#### **stack-trace**

The handler used for subsequently installed signals outputs a Perl stack trace to `STDERR` and then tries to dump core. This is the default signal handler.

**die** The handler used for subsequently installed signals calls `die` (actually `croak`) with a message indicating which signal was caught.

#### **handler *your-handler***

*your-handler* will be used as the handler for subsequently installed signals. *your-handler* can be any value which is valid as an assignment to an element of `%SIG`.

### SIGNAL LISTS

**sigtrap** has a few built-in lists of signals to trap. They are:

#### **normal-signals**

These are the signals which a program might normally expect to encounter and which by default cause it to terminate. They are `HUP`, `INT`, `PIPE` and `TERM`.

#### **error-signals**

These signals usually indicate a serious problem with the Perl interpreter or with your script. They are `ABRT`, `BUS`, `EMT`, `FPE`, `ILL`, `QUIT`, `SEGV`, `SYS` and `TRAP`.

#### **old-interface-signals**

These are the signals which were trapped by default by the old **sigtrap** interface, they are `ABRT`, `BUS`, `EMT`, `FPE`, `ILL`, `PIPE`, `QUIT`, `SEGV`, `SYS`, `TERM`, and `TRAP`. If no signals or signals lists are passed to **sigtrap**, this list is used.

For each of these three lists, the collection of signals set to be trapped is checked before trapping; if your architecture does not implement a particular signal, it will not be trapped but rather silently ignored.

## OTHER

### untrapped

This token tells **sigtrap** to install handlers only for subsequently listed signals which aren't already trapped or ignored.

**any** This token tells **sigtrap** to install handlers for all subsequently listed signals. This is the default behavior.

### signal

Any argument which looks like a signal name (that is, `/^[A-Z] [A-Z0-9] *$/`) indicates that **sigtrap** should install a handler for that name.

### number

Require that at least version *number* of **sigtrap** is being used.

## EXAMPLES

Provide a stack trace for the old-interface-signals:

```
use sigtrap;
```

Ditto:

```
use sigtrap qw(stack-trace old-interface-signals);
```

Provide a stack trace on the 4 listed signals only:

```
use sigtrap qw(BUS SEGV PIPE ABRT);
```

Die on INT or QUIT:

```
use sigtrap qw(die INT QUIT);
```

Die on HUP, INT, PIPE or TERM:

```
use sigtrap qw(die normal-signals);
```

Die on HUP, INT, PIPE or TERM, except don't change the behavior for signals which are already trapped or ignored:

```
use sigtrap qw(die untrapped normal-signals);
```

Die on receipt one of an of the **normal-signals** which is currently **untrapped**, provide a stack trace on receipt of **any** of the **error-signals**:

```
use sigtrap qw(die untrapped normal-signals
 stack-trace any error-signals);
```

Install `my_handler()` as the handler for the **normal-signals**:

```
use sigtrap 'handler', \&my_handler, 'normal-signals';
```

Install `my_handler()` as the handler for the normal-signals, provide a Perl stack trace on receipt of one of the error-signals:

```
use sigtrap qw(handler my_handler normal-signals
 stack-trace error-signals);
```

**NAME**

Socket, sockaddr\_in, sockaddr\_un, inet\_aton, inet\_ntoa – load the C socket.h defines and structure manipulators

**SYNOPSIS**

```
use Socket;

$proto = getprotobyname('udp');
socket(Socket_Handle, PF_INET, SOCK_DGRAM, $proto);
$iaddr = gethostbyname('hishost.com');
$port = getservbyname('time', 'udp');
$sin = sockaddr_in($port, $iaddr);
send(Socket_Handle, 0, 0, $sin);

$proto = getprotobyname('tcp');
socket(Socket_Handle, PF_INET, SOCK_STREAM, $proto);
$port = getservbyname('smtp', 'tcp');
$sin = sockaddr_in($port, inet_aton("127.1"));
$sin = sockaddr_in(7, inet_aton("localhost"));
$sin = sockaddr_in(7, INADDR_LOOPBACK);
connect(Socket_Handle, $sin);

($port, $iaddr) = sockaddr_in(getpeername(Socket_Handle));
$peer_host = gethostbyaddr($iaddr, AF_INET);
$peer_addr = inet_ntoa($iaddr);

$proto = getprotobyname('tcp');
socket(Socket_Handle, PF_UNIX, SOCK_STREAM, $proto);
unlink('/tmp/usock');
$sun = sockaddr_un('/tmp/usock');
connect(Socket_Handle, $sun);
```

**DESCRIPTION**

This module is just a translation of the C *socket.h* file. Unlike the old mechanism of requiring a translated *socket.ph* file, this uses the **h2xs** program (see the Perl source distribution) and your native C compiler. This means that it has a far more likely chance of getting the numbers right. This includes all of the commonly used pound–defines like AF\_INET, SOCK\_STREAM, etc.

Also, some common socket "newline" constants are provided: the constants CR, LF, and CRLF, as well as \$CR, \$LF, and \$CRLF, which map to \015, \012, and \015\012. If you do not want to use the literal characters in your programs, then use the constants provided here. They are not exported by default, but can be imported individually, and with the :crlf export tag:

```
use Socket qw(:DEFAULT :crlf);
```

In addition, some structure manipulation functions are available:

**inet\_aton HOSTNAME**

Takes a string giving the name of a host, and translates that to an opaque string (if programming in C, struct in\_addr). Takes arguments of both the 'rtfm.mit.edu' type and '18.181.0.24'. If the host name cannot be resolved, returns undef. For multi–homed hosts (hosts with more than one address), the first address found is returned.

For portability do not assume that the result of `inet_aton()` is 32 bits wide, in other words, that it would contain only the IPv4 address in network order.

**inet\_ntoa IP\_ADDRESS**

Takes a string (an opaque string as returned by `inet_aton()`, or a v–string representing the four octets of the IPv4 address in network order) and translates it into a string of the form 'd.d.d.d' where

the 'd's are numbers less than 256 (the normal human-readable four dotted number notation for Internet addresses).

#### INADDR\_ANY

Note: does not return a number, but a packed string.

Returns the 4-byte wildcard ip address which specifies any of the hosts ip addresses. (A particular machine can have more than one ip address, each address corresponding to a particular network interface. This wildcard address allows you to bind to all of them simultaneously.) Normally equivalent to `inet_aton('0.0.0.0')`.

#### INADDR\_BROADCAST

Note: does not return a number, but a packed string.

Returns the 4-byte 'this-lan' ip broadcast address. This can be useful for some protocols to solicit information from all servers on the same LAN cable. Normally equivalent to `inet_aton('255.255.255.255')`.

#### INADDR\_LOOPBACK

Note – does not return a number.

Returns the 4-byte loopback address. Normally equivalent to `inet_aton('localhost')`.

#### INADDR\_NONE

Note – does not return a number.

Returns the 4-byte 'invalid' ip address. Normally equivalent to `inet_aton('255.255.255.255')`.

#### sockaddr\_family SOCKADDR

Takes a sockaddr structure (as returned by `pack_sockaddr_in()`, `pack_sockaddr_un()` or the perl builtin functions `getsockname()` and `getpeername()`) and returns the address family tag. It will match the constant `AF_INET` for a `sockaddr_in` and `AF_UNIX` for a `sockaddr_un`. It can be used to figure out what unpacker to use for a sockaddr of unknown type.

#### sockaddr\_in PORT, ADDRESS

##### sockaddr\_in SOCKADDR\_IN

In a list context, unpacks its `SOCKADDR_IN` argument and returns an array consisting of (PORT, ADDRESS). In a scalar context, packs its (PORT, ADDRESS) arguments as a `SOCKADDR_IN` and returns it. If this is confusing, use `pack_sockaddr_in()` and `unpack_sockaddr_in()` explicitly.

#### pack\_sockaddr\_in PORT, IP\_ADDRESS

Takes two arguments, a port number and an opaque string, `IP_ADDRESS` (as returned by `inet_aton()`, or a v-string). Returns the `sockaddr_in` structure with those arguments packed in with `AF_INET` filled in. For Internet domain sockets, this structure is normally what you need for the arguments in `bind()`, `connect()`, and `send()`, and is also returned by `getpeername()`, `getsockname()` and `recv()`.

#### unpack\_sockaddr\_in SOCKADDR\_IN

Takes a `sockaddr_in` structure (as returned by `pack_sockaddr_in()`) and returns an array of two elements: the port and an opaque string representing the IP address (you can use `inet_ntoa()` to convert the address to the four-dotted numeric format). Will croak if the structure does not have `AF_INET` in the right place.

#### sockaddr\_un PATHNAME

##### sockaddr\_un SOCKADDR\_UN

In a list context, unpacks its `SOCKADDR_UN` argument and returns an array consisting of (PATHNAME). In a scalar context, packs its `PATHNAME` arguments as a `SOCKADDR_UN` and returns it. If this is confusing, use `pack_sockaddr_un()` and `unpack_sockaddr_un()`



explicitly. These are only supported if your system has `<sys/un.h>`.

`pack_sockaddr_un` PATH

Takes one argument, a pathname. Returns the `sockaddr_un` structure with that path packed in with `AF_UNIX` filled in. For unix domain sockets, this structure is normally what you need for the arguments in `bind()`, `connect()`, and `send()`, and is also returned by `getpeername()`, `getsockname()` and `recv()`.

`unpack_sockaddr_un` SOCKADDR\_UN

Takes a `sockaddr_un` structure (as returned by `pack_sockaddr_un()`) and returns the pathname. Will croak if the structure does not have `AF_UNIX` in the right place.

**NAME**

sort – perl pragma to control sort () behaviour

**SYNOPSIS**

```
use sort 'stable'; # guarantee stability
use sort '_quicksort'; # use a quicksort algorithm
use sort '_mergesort'; # use a mergesort algorithm

use sort '_qsort'; # alias for quicksort

my $current = sort::current(); # identify prevailing algorithm
```

**DESCRIPTION**

With the sort pragma you can control the behaviour of the builtin sort () function.

In Perl versions 5.6 and earlier the quicksort algorithm was used to implement sort (), but in Perl 5.8 a mergesort algorithm was also made available, mainly to guarantee worst case  $O(N \log N)$  behaviour: the worst case of quicksort is  $O(N^2)$ . In Perl 5.8 and later, quicksort defends against quadratic behaviour by shuffling large arrays before sorting.

A stable sort means that for records that compare equal, the original input ordering is preserved. Mergesort is stable, quicksort is not. Stability will matter only if elements that compare equal can be distinguished in some other way. That means that simple numerical and lexical sorts do not profit from stability, since equal elements are indistinguishable. However, with a comparison such as

```
{ substr($a, 0, 3) cmp substr($b, 0, 3) }
```

stability might matter because elements that compare equal on the first 3 characters may be distinguished based on subsequent characters. In Perl 5.8 and later, quicksort can be stabilized, but doing so will add overhead, so it should only be done if it matters.

The best algorithm depends on many things. On average, mergesort does fewer comparisons than quicksort, so it may be better when complicated comparison routines are used. Mergesort also takes advantage of pre-existing order, so it would be favored for using sort to merge several sorted arrays. On the other hand, quicksort is often faster for small arrays, and on platforms with small memory caches that are much faster than main memory. You can force the choice of algorithm with this pragma, but this feels heavy-handed, so the subpragmas beginning with a \_ may not persist beyond Perl 5.8.

**CAVEATS**

This pragma is not lexically scoped : its effect is global to the program it appears in. This may change in future versions.

**NAME**

Storable – persistence for Perl data structures

**SYNOPSIS**

```
use Storable;
store \%table, 'file';
$hashref = retrieve('file');

use Storable qw(nstore store_fd nstore_fd freeze thaw dclone);

Network order
nstore \%table, 'file';
$hashref = retrieve('file'); # There is NO nretrieve()

Storing to and retrieving from an already opened file
store_fd \@array, *STDOUT;
nstore_fd \%table, *STDOUT;
$arrayref = fd_retrieve(*SOCKET);
$hashref = fd_retrieve(*SOCKET);

Serializing to memory
$serialized = freeze \%table;
%table_clone = %{ thaw($serialized) };

Deep (recursive) cloning
$cloneref = dclone($ref);

Advisory locking
use Storable qw(lock_store lock_nstore lock_retrieve)
lock_store \%table, 'file';
lock_nstore \%table, 'file';
$hashref = lock_retrieve('file');
```

**DESCRIPTION**

The Storable package brings persistence to your Perl data structures containing SCALAR, ARRAY, HASH or REF objects, i.e. anything that can be conveniently stored to disk and retrieved at a later time.

It can be used in the regular procedural way by calling `store` with a reference to the object to be stored, along with the file name where the image should be written.

The routine returns `undef` for I/O problems or other internal error, a true value otherwise. Serious errors are propagated as a die exception.

To retrieve data stored to disk, use `retrieve` with a file name. The objects stored into that file are recreated into memory for you, and a *reference* to the root object is returned. In case an I/O error occurs while reading, `undef` is returned instead. Other serious errors are propagated via `die`.

Since storage is performed recursively, you might want to stuff references to objects that share a lot of common data into a single array or hash table, and then store that object. That way, when you retrieve back the whole thing, the objects will continue to share what they originally shared.

At the cost of a slight header overhead, you may store to an already opened file descriptor using the `store_fd` routine, and retrieve from a file via `fd_retrieve`. Those names aren't imported by default, so you will have to do that explicitly if you need those routines. The file descriptor you supply must be already opened, for read if you're going to retrieve and for write if you wish to store.

```
store_fd(\%table, *STDOUT) || die "can't store to stdout\n";
$hashref = fd_retrieve(*STDIN);
```

You can also store data in network order to allow easy sharing across multiple platforms, or when storing on a socket known to be remotely connected. The routines to call have an initial `n` prefix for *network*, as in

`nstore` and `nstore_fd`. At retrieval time, your data will be correctly restored so you don't have to know whether you're restoring from native or network ordered data. Double values are stored stringified to ensure portability as well, at the slight risk of loosing some precision in the last decimals.

When using `fd_retrieve`, objects are retrieved in sequence, one object (i.e. one recursive tree) per associated `store_fd`.

If you're more from the object-oriented camp, you can inherit from `Storable` and directly store your objects by invoking `store` as a method. The fact that the root of the to-be-stored tree is a blessed reference (i.e. an object) is special-cased so that the retrieve does not provide a reference to that object but rather the blessed object reference itself. (Otherwise, you'd get a reference to that blessed object).

## MEMORY STORE

The `Storable` engine can also store data into a Perl scalar instead, to later retrieve them. This is mainly used to freeze a complex structure in some safe compact memory place (where it can possibly be sent to another process via some IPC, since freezing the structure also serializes it in effect). Later on, and maybe somewhere else, you can thaw the Perl scalar out and recreate the original complex structure in memory.

Surprisingly, the routines to be called are named `freeze` and `thaw`. If you wish to send out the frozen scalar to another machine, use `nfreeze` instead to get a portable image.

Note that freezing an object structure and immediately thawing it actually achieves a deep cloning of that structure:

```
dclone(.) = thaw(freeze(.))
```

`Storable` provides you with a `dclone` interface which does not create that intermediary scalar but instead freezes the structure in some internal memory space and then immediately thaws it out.

## ADVISORY LOCKING

The `lock_store` and `lock_nstore` routine are equivalent to `store` and `nstore`, except that they get an exclusive lock on the file before writing. Likewise, `lock_retrieve` does the same as `retrieve`, but also gets a shared lock on the file before reading.

As with any advisory locking scheme, the protection only works if you systematically use `lock_store` and `lock_retrieve`. If one side of your application uses `store` whilst the other uses `lock_retrieve`, you will get no protection at all.

The internal advisory locking is implemented using Perl's `flock()` routine. If your system does not support any form of `flock()`, or if you share your files across NFS, you might wish to use other forms of locking by using modules such as `LockFile::Simple` which lock a file using a filesystem entry, instead of locking the file descriptor.

## SPEED

The heart of `Storable` is written in C for decent speed. Extra low-level optimizations have been made when manipulating perl internals, to sacrifice encapsulation for the benefit of greater speed.

## CANONICAL REPRESENTATION

Normally, `Storable` stores elements of hashes in the order they are stored internally by Perl, i.e. pseudo-randomly. If you set `$Storable::canonical` to some TRUE value, `Storable` will store hashes with the elements sorted by their key. This allows you to compare data structures by comparing their frozen representations (or even the compressed frozen representations), which can be useful for creating lookup tables for complicated queries.

Canonical order does not imply network order; those are two orthogonal settings.

## FORWARD COMPATIBILITY

This release of `Storable` can be used on a newer version of Perl to serialize data which is not supported by earlier Perls. By default, `Storable` will attempt to do the right thing, by `croak()`ing if it encounters data that it cannot deserialize. However, the defaults can be changed as follows:

### utf8 data

Perl 5.6 added support for Unicode characters with code points `> 255`, and Perl 5.8 has full support for Unicode characters in hash keys. Perl internally encodes strings with these characters using utf8, and Storable serializes them as utf8. By default, if an older version of Perl encounters a utf8 value it cannot represent, it will `croak()`. To change this behaviour so that Storable deserializes utf8 encoded values as the string of bytes (effectively dropping the `is_utf8` flag) set `$Storable::drop_utf8` to some TRUE value. This is a form of data loss, because with `$drop_utf8` true, it becomes impossible to tell whether the original data was the Unicode string, or a series of bytes that happen to be valid utf8.

### restricted hashes

Perl 5.8 adds support for restricted hashes, which have keys restricted to a given set, and can have values locked to be read only. By default, when Storable encounters a restricted hash on a perl that doesn't support them, it will deserialize it as a normal hash, silently discarding any placeholder keys and leaving the keys and all values unlocked. To make Storable `croak()` instead, set `$Storable::downgrade_restricted` to a FALSE value. To restore the default set it back to some TRUE value.

### files from future versions of Storable

Earlier versions of Storable would immediately croak if they encountered a file with a higher internal version number than the reading Storable knew about. Internal version numbers are increased each time new data types (such as restricted hashes) are added to the vocabulary of the file format. This meant that a newer Storable module had no way of writing a file readable by an older Storable, even if the writer didn't store newer data types.

This version of Storable will defer croaking until it encounters a data type in the file that it does not recognize. This means that it will continue to read files generated by newer Storable modules which are careful in what they write out, making it easier to upgrade Storable modules in a mixed environment.

The old behaviour of immediate croaking can be re-instated by setting `$Storable::accept_future_minor` to some FALSE value.

All these variables have no effect on a newer Perl which supports the relevant feature.

## ERROR REPORTING

Storable uses the "exception" paradigm, in that it does not try to workaround failures: if something bad happens, an exception is generated from the caller's perspective (see [Carp](#) and `croak()`). Use `eval {}` to trap those exceptions.

When Storable croaks, it tries to report the error via the `logcroak()` routine from the `Log::Agent` package, if it is available.

Normal errors are reported by having `store()` or `retrieve()` return `undef`. Such errors are usually I/O errors (or truncated stream errors at retrieval).

## WIZARDS ONLY

### Hooks

Any class may define hooks that will be called during the serialization and deserialization process on objects that are instances of that class. Those hooks can redefine the way serialization is performed (and therefore, how the symmetrical deserialization should be conducted).

Since we said earlier:

```
dclone(.) = thaw(freeze(.))
```

everything we say about hooks should also hold for deep cloning. However, hooks get to know whether the operation is a mere serialization, or a cloning.

Therefore, when serializing hooks are involved,

```
dclone(.) <> thaw(freeze(.))
```

Well, you could keep them in sync, but there's no guarantee it will always hold on classes somebody else wrote. Besides, there is little to gain in doing so: a serializing hook could keep only one attribute of an object, which is probably not what should happen during a deep cloning of that same object.

Here is the hooking interface:

`STORABLE_freeze` *obj, cloning*

The serializing hook, called on the object during serialization. It can be inherited, or defined in the class itself, like any other method.

Arguments: *obj* is the object to serialize, *cloning* is a flag indicating whether we're in a `dclone()` or a regular serialization via `store()` or `freeze()`.

Returned value: A LIST (`$serialized`, `$ref1`, `$ref2`, ...) where `$serialized` is the serialized form to be used, and the optional `$ref1`, `$ref2`, etc... are extra references that you wish to let the Storable engine serialize.

At deserialization time, you will be given back the same LIST, but all the extra references will be pointing into the deserialized structure.

The **first time** the hook is hit in a serialization flow, you may have it return an empty list. That will signal the Storable engine to further discard that hook for this class and to therefore revert to the default serialization of the underlying Perl data. The hook will again be normally processed in the next serialization.

Unless you know better, serializing hook should always say:

```
sub STORABLE_freeze {
 my ($self, $cloning) = @_;
 return if $cloning; # Regular default serialization

}
```

in order to keep reasonable `dclone()` semantics.

`STORABLE_thaw` *obj, cloning, serialized, ...*

The deserializing hook called on the object during deserialization. But wait: if we're deserializing, there's no object yet... right?

Wrong: the Storable engine creates an empty one for you. If you know Eiffel, you can view `STORABLE_thaw` as an alternate creation routine.

This means the hook can be inherited like any other method, and that *obj* is your blessed reference for this particular instance.

The other arguments should look familiar if you know `STORABLE_freeze`: *cloning* is true when we're part of a deep clone operation, *serialized* is the serialized string you returned to the engine in `STORABLE_freeze`, and there may be an optional list of references, in the same order you gave them at serialization time, pointing to the deserialized objects (which have been processed courtesy of the Storable engine).

When the Storable engine does not find any `STORABLE_thaw` hook routine, it tries to load the class by requiring the package dynamically (using the blessed package name), and then re-attempts the lookup. If at that time the hook cannot be located, the engine croaks. Note that this mechanism will fail if you define several classes in the same file, but *perlmod* warned you.

It is up to you to use this information to populate *obj* the way you want.

Returned value: none.

## Predicates

Predicates are not exportable. They must be called by explicitly prefixing them with the Storable package name.

`Storable::last_op_in_netorder`

The `Storable::last_op_in_netorder()` predicate will tell you whether network order was used in the last store or retrieve operation. If you don't know how to use this, just forget about it.

`Storable::is_storing`

Returns true if within a store operation (via `STORABLE_freeze` hook).

`Storable::is_retrieving`

Returns true if within a retrieve operation (via `STORABLE_thaw` hook).

## Recursion

With hooks comes the ability to recurse back to the Storable engine. Indeed, hooks are regular Perl code, and Storable is convenient when it comes to serializing and deserializing things, so why not use it to handle the serialization string?

There are a few things you need to know, however:

- You can create endless loops if the things you serialize via `freeze()` (for instance) point back to the object we're trying to serialize in the hook.
- Shared references among objects will not stay shared: if we're serializing the list of object [A, C] where both object A and C refer to the SAME object B, and if there is a serializing hook in A that says `freeze(B)`, then when deserializing, we'll get [A', C'] where A' refers to B', but C' refers to D, a deep clone of B'. The topology was not preserved.

That's why `STORABLE_freeze` lets you provide a list of references to serialize. The engine guarantees that those will be serialized in the same context as the other objects, and therefore that shared objects will stay shared.

In the above [A, C] example, the `STORABLE_freeze` hook could return:

```
("something", $self->{B})
```

and the B part would be serialized by the engine. In `STORABLE_thaw`, you would get back the reference to the B' object, deserialized for you.

Therefore, recursion should normally be avoided, but is nonetheless supported.

## Deep Cloning

There is a Clone module available on CPAN which implements deep cloning natively, i.e. without freezing to memory and thawing the result. It is aimed to replace Storable's `dclone()` some day. However, it does not currently support Storable hooks to redefine the way deep cloning is performed.

## Storable magic

Yes, there's a lot of that :-). But more precisely, in UNIX systems there's a utility called `file`, which recognizes data files based on their contents (usually their first few bytes). For this to work, a certain file called *magic* needs to be taught about the *signature* of the data. Where that configuration file lives depends on the UNIX flavour; often it's something like `/usr/share/misc/magic` or `/etc/magic`. Your system administrator needs to do the updating of the *magic* file. The necessary signature information is output to STDOUT by invoking `Storable::show_file_magic()`. Note that the GNU implementation of the `file` utility, version 3.38 or later, is expected to contain support for recognising Storable files out-of-the-box, in addition to other kinds of Perl files.

## EXAMPLES

Here are some code samples showing a possible usage of Storable:

```

use Storable qw(store retrieve freeze thaw dclone);

%color = ('Blue' => 0.1, 'Red' => 0.8, 'Black' => 0, 'White' => 1);
store(\%color, '/tmp/colors') or die "Can't store %a in /tmp/colors!\n";
$colref = retrieve('/tmp/colors');
die "Unable to retrieve from /tmp/colors!\n" unless defined $colref;
printf "Blue is still %lf\n", $colref->{'Blue'};

$colref2 = dclone(\%color);

$str = freeze(\%color);
printf "Serialization of %color is %d bytes long.\n", length($str);
$colref3 = thaw($str);

```

which prints (on my machine):

```

Blue is still 0.100000
Serialization of %color is 102 bytes long.

```

## WARNING

If you're using references as keys within your hash tables, you're bound to be disappointed when retrieving your data. Indeed, Perl stringifies references used as hash table keys. If you later wish to access the items via another reference stringification (i.e. using the same reference that was used for the key originally to record the value into the hash table), it will work because both references stringify to the same string.

It won't work across a sequence of `store` and `retrieve` operations, however, because the addresses in the retrieved objects, which are part of the stringified references, will probably differ from the original addresses. The topology of your structure is preserved, but not hidden semantics like those.

On platforms where it matters, be sure to call `binmode()` on the descriptors that you pass to Storable functions.

Storing data canonically that contains large hashes can be significantly slower than storing the same data normally, as temporary arrays to hold the keys for each hash have to be allocated, populated, sorted and freed. Some tests have shown a halving of the speed of storing — the exact penalty will depend on the complexity of your data. There is no slowdown on retrieval.

## BUGS

You can't store `GLOB`, `CODE`, `FORMLINE`, etc.... If you can define semantics for those operations, feel free to enhance Storable so that it can deal with them.

The store functions will croak if they run into such references unless you set `$Storable::forgive_me` to some TRUE value. In that case, the fatal message is turned in a warning and some meaningless string is stored instead.

Setting `$Storable::canonical` may not yield frozen strings that compare equal due to possible stringification of numbers. When the string version of a scalar exists, it is the form stored; therefore, if you happen to use your numbers as strings between two freezing operations on the same data structures, you will get different results.

When storing doubles in network order, their value is stored as text. However, you should also not expect non-numeric floating-point values such as infinity and "not a number" to pass successfully through a `nstore()/retrieve()` pair.

As Storable neither knows nor cares about character sets (although it does know that characters may be more than eight bits wide), any difference in the interpretation of character codes between a host and a target system is your problem. In particular, if host and target use different code points to represent the characters



used in the text representation of floating-point numbers, you will not be able to exchange floating-point data, even with `nstore()`.

`Storable::drop_utf8` is a blunt tool. There is no facility either to return **all** strings as utf8 sequences, or to attempt to convert utf8 data back to 8 bit and `croak()` if the conversion fails.

Prior to Storable 2.01, no distinction was made between signed and unsigned integers on storing. By default Storable prefers to store a scalar's string representation (if it has one) so this would only cause problems when storing large unsigned integers that had never been converted to string or floating point. In other words values that had been generated by integer operations such as logic ops and then not used in any string or arithmetic context before storing.

### 64 bit data in perl 5.6.0 and 5.6.1

This section only applies to you if you have existing data written out by Storable 2.02 or earlier on perl 5.6.0 or 5.6.1 on Unix or Linux which has been configured with 64 bit integer support (not the default). If you got a precompiled perl, rather than running Configure to build your own perl from source, then it almost certainly does not affect you, and you can stop reading now (unless you're curious). If you're using perl on Windows it does not affect you.

Storable writes a file header which contains the sizes of various C language types for the C compiler that built Storable (when not writing in network order), and will refuse to load files written by a Storable not on the same (or compatible) architecture. This check and a check on machine byteorder is needed because the size of various fields in the file are given by the sizes of the C language types, and so files written on different architectures are incompatible. This is done for increased speed. (When writing in network order, all fields are written out as standard lengths, which allows full interworking, but takes longer to read and write)

Perl 5.6.x introduced the ability to optionally configure the perl interpreter to use C's `long long` type to allow scalars to store 64 bit integers on 32 bit systems. However, due to the way the Perl configuration system generated the C configuration files on non-Windows platforms, and the way Storable generates its header, nothing in the Storable file header reflected whether the perl writing was using 32 or 64 bit integers, despite the fact that Storable was storing some data differently in the file. Hence Storable running on perl with 64 bit integers will read the header from a file written by a 32 bit perl, not realise that the data is actually in a subtly incompatible format, and then go horribly wrong (possibly crashing) if it encountered a stored integer. This is a design failure.

Storable has now been changed to write out and read in a file header with information about the size of integers. It's impossible to detect whether an old file being read in was written with 32 or 64 bit integers (they have the same header) so it's impossible to automatically switch to a correct backwards compatibility mode. Hence this Storable defaults to the new, correct behaviour.

What this means is that if you have data written by Storable 1.x running on perl 5.6.0 or 5.6.1 configured with 64 bit integers on Unix or Linux then by default this Storable will refuse to read it, giving the error *Byte order is not compatible*. If you have such data then you should set `$Storable::interwork_56_64bit` to a true value to make this Storable read and write files with the old header. You should also migrate your data, or any older perl you are communicating with, to this current version of Storable.

If you don't have data written with specific configuration of perl described above, then you do not and should not do anything. Don't set the flag – not only will Storable on an identically configured perl refuse to load them, but Storable a differently configured perl will load them believing them to be correct for it, and then may well fail or crash part way through reading them.

### CREDITS

Thank you to (in chronological order):

Jarkko Hietaniemi <jhi@iki.fi>  
Ulrich Pfeifer <pfeifer@charly.informatik.uni-dortmund.de>  
Benjamin A. Holzman <bah@ecnavantage.com>

Andrew Ford <A.Ford@ford-mason.co.uk>  
Gisle Aas <gisle@aas.no>  
Jeff Gresham <gresham\_jeffrey@jpmorgan.com>  
Murray Nesbitt <murray@activestate.com>  
Marc Lehmann <pcg@opengroup.org>  
Justin Banks <justinb@wamnet.com>  
Jarkko Hietaniemi <jhi@iki.fi> (AGAIN, as perl 5.7.0 Pumpkin!)  
Salvador Ortiz Garcia <sog@msg.com.mx>  
Dominic Dunlop <domo@computer.org>  
Erik Haugan <erik@solbors.no>

for their bug reports, suggestions and contributions.

Benjamin Holzman contributed the tied variable support, Andrew Ford contributed the canonical order for hashes, and Gisle Aas fixed a few misunderstandings of mine regarding the perl internals, and optimized the emission of "tags" in the output streams by simply counting the objects instead of tagging them (leading to a binary incompatibility for the Storable image starting at version 0.6—older images are, of course, still properly understood). Murray Nesbitt made Storable thread-safe. Marc Lehmann added overloading and references to tied items support.

## AUTHOR

Storable was written by Raphael Manfredi <*Raphael\_Manfredi@pobox.com*> Maintenance is now done by the perl5-porters <*perl5-porters@perl.org*>

Please e-mail us with problems, bug fixes, comments and complaints, although if you have complements you should send them to Raphael. Please don't e-mail Raphael with problems, as he no longer works on Storable, and your message will be delayed while he forwards it to us.

## SEE ALSO

[\*Clone\*](#).

**NAME**

strict – Perl pragma to restrict unsafe constructs

**SYNOPSIS**

```
use strict;

use strict "vars";
use strict "refs";
use strict "subs";

use strict;
no strict "vars";
```

**DESCRIPTION**

If no import list is supplied, all possible restrictions are assumed. (This is the safest mode to operate in, but is sometimes too strict for casual programming.) Currently, there are three possible things to be strict about: "subs", "vars", and "refs".

strict refs

This generates a runtime error if you use symbolic references (see [perlref](#)).

```
use strict 'refs';
$ref = \ $foo;
print $$ref; # ok
$ref = "foo";
print $$ref; # runtime error; normally ok
$file = "STDOUT";
print $file "Hi!"; # error; note: no comma after $file
```

There is one exception to this rule:

```
$bar = \&{'foo'};
&$bar;
```

is allowed so that `goto &$AUTOLOAD` would not break under stricture.

strict vars

This generates a compile-time error if you access a variable that wasn't declared via `"our"` or `use vars`, localized via `my()`, or wasn't fully qualified. Because this is to avoid variable suicide problems and subtle dynamic scoping issues, a merely `local()` variable isn't good enough. See [my](#) and [local](#).

```
use strict 'vars';
$X::foo = 1; # ok, fully qualified
my $foo = 10; # ok, my() var
local $foo = 9; # blows up

package Cinna;
our $bar; # Declares $bar in current package
$bar = 'HgS'; # ok, global declared via pragma
```

The `local()` generated a compile-time error because you just touched a global name without fully qualifying it.

Because of their special use by `sort()`, the variables `$a` and `$b` are exempted from this check.

strict subs

This disables the poetry optimization, generating a compile-time error if you try to use a bareword identifier that's not a subroutine, unless it appears in curly braces or on the left hand side of the `"=>"` symbol.

```
use strict 'subs';
$SIG{PIPE} = Plumber# blows up
$SIG{PIPE} = "Plumbe#" just fine: bareword in curlies always ok
$SIG{PIPE} = \&Plumber# preferred form
```

See [Pragmatic Modules](#).

**NAME**

subs – Perl pragma to predeclare sub names

**SYNOPSIS**

```
use subs qw(frob);
frob 3..10;
```

**DESCRIPTION**

This will predeclare all the subroutine whose names are in the list, allowing you to use them without parentheses even before they're declared.

Unlike pragmas that affect the `%^H` hints variable, the `use vars` and `use subs` declarations are not BLOCK-scoped. They are thus effective for the entire file in which they appear. You may not rescind such declarations with `no vars` or `no subs`.

See [Pragmatic Modules](#) and [strict subs](#).

**NAME**

Switch – A switch statement for Perl

**VERSION**

This document describes version 2.09 of Switch, released June 12, 2002.

**SYNOPSIS**

```
use Switch;

switch ($val) {

 case 1 { print "number 1" }
 case "a" { print "string a" }
 case [1..10,42] { print "number in list" }
 case (@array) { print "number in list" }
 case /\w+/ { print "pattern" }
 case qr/\w+/ { print "pattern" }
 case (%hash) { print "entry in hash" }
 case (\%hash) { print "entry in hash" }
 case (\&sub) { print "arg to subroutine" }
 else { print "previous case not true" }

}
```

**BACKGROUND**

[Skip ahead to "[DESCRIPTION](#)" if you don't care about the whys and wherefores of this control structure]

In seeking to devise a "Swiss Army" case mechanism suitable for Perl, it is useful to generalize this notion of distributed conditional testing as far as possible. Specifically, the concept of "matching" between the switch value and the various case values need not be restricted to numeric (or string or referential) equality, as it is in other languages. Indeed, as Table 1 illustrates, Perl offers at least eighteen different ways in which two values could generate a match.

Table 1: Matching a switch value (\$s) with a case value (\$c)

| Switch Value         | Case Value           | Type of Match Implied                                                                       | Matching Code                                                                        |
|----------------------|----------------------|---------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| =====                | =====                | =====                                                                                       | =====                                                                                |
| number or ref        | same                 | numeric or referential equality                                                             | match if \$s == \$c;                                                                 |
| object ref           | method name or ref   | result of method call                                                                       | match if \$s->\$c();<br>match if defined \$s->\$c();                                 |
| other non-ref scalar | other non-ref scalar | string equality                                                                             | match if \$s eq \$c;                                                                 |
| string               | regex                | pattern match                                                                               | match if \$s =~ /\$c/;                                                               |
| array ref            | scalar               | array entry existence<br>array entry definition<br>array entry truth                        | match if 0<=\$c && \$c<@\$s;<br>match if defined \$s->[\$c];<br>match if \$s->[\$c]; |
| array ref            | array ref            | array intersection<br>(apply this table to all pairs of elements \$s->[\$i] and \$c->[\$j]) | match if intersects(@\$s, @\$c);                                                     |

|              |        |                                                                   |                                                                                     |
|--------------|--------|-------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| array<br>ref | regex  | array grep                                                        | match if grep /\$c/, @\$s;                                                          |
| hash<br>ref  | scalar | hash entry existence<br>hash entry definition<br>hash entry truth | match if exists \$s->{\$c};<br>match if defined \$s->{\$c};<br>match if \$s->{\$c}; |
| hash<br>ref  | regex  | hash grep                                                         | match if grep /\$c/, keys %\$s;                                                     |
| sub<br>ref   | scalar | return value defn<br>return value truth                           | match if defined \$s->(\$c);<br>match if \$s->(\$c);                                |
| sub<br>ref   | array  | return value defn<br>return value truth                           | match if defined \$s->(@\$c);<br>match if \$s->(@\$c);                              |

In reality, Table 1 covers 31 alternatives, because only the equality and intersection tests are commutative; in all other cases, the roles of the `$s` and `$c` variables could be reversed to produce a different test. For example, instead of testing a single hash for the existence of a series of keys (`match if exists $s->{$c}`), one could test for the existence of a single key in a series of hashes (`match if exists $c->{$s}`).

As *perlto* observes, a Perl case mechanism must support all these "ways to do it".

## DESCRIPTION

The `Switch.pm` module implements a generalized case mechanism that covers the numerous possible combinations of switch and case values described above.

The module augments the standard Perl syntax with two new control statements: `switch` and `case`. The `switch` statement takes a single scalar argument of any type, specified in parentheses. `switch` stores this value as the current switch value in a (localized) control variable. The value is followed by a block which may contain one or more Perl statements (including the `case` statement described below). The block is unconditionally executed once the switch value has been cached.

A `case` statement takes a single scalar argument (in mandatory parentheses if it's a variable; otherwise the parens are optional) and selects the appropriate type of matching between that argument and the current switch value. The type of matching used is determined by the respective types of the switch value and the case argument, as specified in Table 1. If the match is successful, the mandatory block associated with the case statement is executed.

In most other respects, the `case` statement is semantically identical to an `if` statement. For example, it can be followed by an `else` clause, and can be used as a postfix statement qualifier.

However, when a `case` block has been executed control is automatically transferred to the statement after the immediately enclosing `switch` block, rather than to the next statement within the block. In other words, the success of any `case` statement prevents other cases in the same scope from executing. But see *"Allowing fall-through"* below.

Together these two new statements provide a fully generalized case mechanism:

```
use Switch;

AND LATER...

%special = (woohoo => 1, d'oh => 1);

while (<>) {
 switch ($_) {
 case (%special) { print "homer\n"; } # if $special{$_}
 case /a-z/i { print "alpha\n"; } # if $_ =~ /a-z/i
 case [1..9] { print "small num\n"; } # if $_ in [1..9]
```

```

 case { $_[0] >= 10 } { # if $_ >= 10
 my $age = <>;
 switch (sub{ $_[0] < $age }) {
 case 20 { print "teens\n"; } # if 20 < $age
 case 30 { print "twenties\n"; } # if 30 < $age
 else { print "history\n"; }
 }
 }
 print "must be punctuation\n" case /\W/; # if $_ =~ /\W/
}

```

Note that switches can be nested within case (or any other) blocks, and a series of case statements can try different types of matches — hash membership, pattern match, array intersection, simple equality, etc. — against the same switch value.

The use of intersection tests against an array reference is particularly useful for aggregating integral cases:

```

sub classify_digit
{
 switch ($_[0]) { case 0 { return 'zero' }
 case [2,4,6,8] { return 'even' }
 case [1,3,4,7,9] { return 'odd' }
 case /[A-F]/i { return 'hex' }
 }
}

```

### Allowing fall-through

Fall-through (trying another case after one has already succeeded) is usually a Bad Idea in a switch statement. However, this is Perl, not a police state, so there *is* a way to do it, if you must.

If a case block executes an untargetted `next`, control is immediately transferred to the statement *after* the case statement (i.e. usually another case), rather than out of the surrounding switch block.

For example:

```

switch ($val) {
 case 1 { handle_num_1(); next } # and try next case...
 case "1" { handle_str_1(); next } # and try next case...
 case [0..9] { handle_num_any(); } # and we're done
 case /\d/ { handle_dig_any(); next } # and try next case...
 case /.*/ { handle_str_any(); next } # and try next case...
}

```

If `$val` held the number 1, the above switch block would call the first three `handle_...` subroutines, jumping to the next case test each time it encountered a `next`. After the third case block was executed, control would jump to the end of the enclosing switch block.

On the other hand, if `$val` held 10, then only the last two `handle_...` subroutines would be called.

Note that this mechanism allows the notion of *conditional fall-through*. For example:

```

switch ($val) {
 case [0..9] { handle_num_any(); next if $val < 7; }
 case /\d/ { handle_dig_any(); }
}

```

If an untargetted last statement is executed in a case block, this immediately transfers control out of the enclosing switch block (in other words, there is an implicit `last` at the end of each normal case block). Thus the previous example could also have been written:



```

switch ($val) {
 case [0..9] { handle_num_any(); last if $val >= 7; next; }
 case /\d/ { handle_dig_any(); }
}

```

### Automating fall-through

In situations where case fall-through should be the norm, rather than an exception, an endless succession of terminal `next`s is tedious and ugly. Hence, it is possible to reverse the default behaviour by specifying the string "fallthrough" when importing the module. For example, the following code is equivalent to the first example in *"Allowing fall-through"*:

```

use Switch 'fallthrough';

switch ($val) {
 case 1 { handle_num_1(); }
 case "1" { handle_str_1(); }
 case [0..9] { handle_num_any(); last }
 case /\d/ { handle_dig_any(); }
 case /.*/ { handle_str_any(); }
}

```

Note the explicit use of a `last` to preserve the non-fall-through behaviour of the third case.

### Alternative syntax

Perl 6 will provide a built-in switch statement with essentially the same semantics as those offered by `Switch.pm`, but with a different pair of keywords. In Perl 6 `switch` will be spelled `given`, and `case` will be pronounced `when`. In addition, the `when` statement will not require switch or case values to be parenthesized.

This future syntax is also (largely) available via the `Switch.pm` module, by importing it with the argument "Perl6". For example:

```

use Switch 'Perl6';

given ($val) {
 when 1 { handle_num_1(); }
 when ($str1) { handle_str_1(); }
 when [0..9] { handle_num_any(); last }
 when /\d/ { handle_dig_any(); }
 when /.*/ { handle_str_any(); }
}

```

Note that scalars still need to be parenthesized, since they would be ambiguous in Perl 5.

Note too that you can mix and match both syntaxes by importing the module with:

```

use Switch 'Perl5', 'Perl6';

```

### Higher-order Operations

One situation in which `switch` and `case` do not provide a good substitute for a cascaded `if`, is where a switch value needs to be tested against a series of conditions. For example:

```

sub beverage {
 switch (shift) {
 case sub { $_[0] < 10 } { return 'milk' }
 case sub { $_[0] < 20 } { return 'coke' }
 case sub { $_[0] < 30 } { return 'beer' }
 case sub { $_[0] < 40 } { return 'wine' }
 case sub { $_[0] < 50 } { return 'malt' }
 case sub { $_[0] < 60 } { return 'Moet' }
 }
}

```

```

 else { return 'milk' }
 }
}

```

The need to specify each condition as a subroutine block is tiresome. To overcome this, when importing Switch.pm, a special "placeholder" subroutine named `__` [sic] may also be imported. This subroutine converts (almost) any expression in which it appears to a reference to a higher-order function. That is, the expression:

```

use Switch '__';

__ < 2 + __

```

is equivalent to:

```

sub { $_[0] < 2 + $_[1] }

```

With `__`, the previous ugly case statements can be rewritten:

```

case __ < 10 { return 'milk' }
case __ < 20 { return 'coke' }
case __ < 30 { return 'beer' }
case __ < 40 { return 'wine' }
case __ < 50 { return 'malt' }
case __ < 60 { return 'Moet' }
else { return 'milk' }

```

The `__` subroutine makes extensive use of operator overloading to perform its magic. All operations involving `__` are overloaded to produce an anonymous subroutine that implements a lazy version of the original operation.

The only problem is that operator overloading does not allow the boolean operators `&&` and `||` to be overloaded. So a case statement like this:

```

case 0 <= __ && __ < 10 { return 'digit' }

```

doesn't act as expected, because when it is executed, it constructs two higher order subroutines and then treats the two resulting references as arguments to `&&`:

```

sub { 0 <= $_[0] } && sub { $_[0] < 10 }

```

This boolean expression is inevitably true, since both references are non-`false`. Fortunately, the overloaded `'bool'` operator catches this situation and flags it as a error.

## DEPENDENCIES

The module is implemented using `Filter::Util::Call` and `Text::Balanced` and requires both these modules to be installed.

## AUTHOR

Damian Conway (damian@conway.org)

## BUGS

There are undoubtedly serious bugs lurking somewhere in code this funky :-). Bug reports and other feedback are most welcome.

## LIMITATION

Due to the heuristic nature of Switch.pm's source parsing, the presence of regexes specified with raw `?...?` delimiters may cause mysterious errors. The workaround is to use `m?...?` instead.

## COPYRIGHT

Copyright (c) 1997-2001, Damian Conway. All Rights Reserved.  
This module is free software. It may be used, redistributed

and/or modified under the same terms as Perl itself.

**NAME**

Symbol – manipulate Perl symbols and their names

**SYNOPSIS**

```
use Symbol;

$sym = gensym;
open($sym, "filename");
$_ = <$sym>;
etc.

ungensym $sym; # no effect

replace *FOO{IO} handle but not $FOO, %FOO, etc.
*FOO = geniosym;

print qualify("x"), "\n"; # "Test::x"
print qualify("x", "FOO"), "\n"; # "FOO::x"
print qualify("BAR::x"), "\n"; # "BAR::x"
print qualify("BAR::x", "FOO"), "\n"; # "BAR::x"
print qualify("STDOUT", "FOO"), "\n"; # "main::STDOUT" (global)
print qualify(*x), "\n"; # returns *x
print qualify(*x, "FOO"), "\n"; # returns *x

use strict refs;
print { qualify_to_ref $fh } "foo!\n";
$ref = qualify_to_ref $name, $pkg;

use Symbol qw(delete_package);
delete_package('Foo::Bar');
print "deleted\n" unless exists $Foo::{'Bar::'};
```

**DESCRIPTION**

`Symbol::gensym` creates an anonymous glob and returns a reference to it. Such a glob reference can be used as a file or directory handle.

For backward compatibility with older implementations that didn't support anonymous globs, `Symbol::ungensym` is also provided. But it doesn't do anything.

`Symbol::geniosym` creates an anonymous IO handle. This can be assigned into an existing glob without affecting the non-IO portions of the glob.

`Symbol::qualify` turns unqualified symbol names into qualified variable names (e.g. "myvar" → "MyPackage::myvar"). If it is given a second parameter, `qualify` uses it as the default package; otherwise, it uses the package of its caller. Regardless, global variable names (e.g. "STDOUT", "ENV", "SIG") are always qualified with "main::".

Qualification applies only to symbol names (strings). References are left unchanged under the assumption that they are glob references, which are qualified by their nature.

`Symbol::qualify_to_ref` is just like `Symbol::qualify` except that it returns a glob ref rather than a symbol name, so you can use the result even if `use strict 'refs'` is in effect.

`Symbol::delete_package` wipes out a whole package namespace. Note this routine is not exported by default—you may want to import it explicitly.

**NAME**

Sys::Hostname – Try every conceivable way to get hostname

**SYNOPSIS**

```
use Sys::Hostname;
$host = hostname;
```

**DESCRIPTION**

Attempts several methods of getting the system hostname and then caches the result. It tries the first available of the C library's `gethostname()`, ``$Config{aphostname}'`, `uname(2)`, `syscall(SYS_gethostname)`, `'hostname'`, `'uname -n'`, and the file */com/host*. If all that fails it croaks.

All NULs, returns, and newlines are removed from the result.

**AUTHOR**

David Sundstrom <*sunds@asictest.sc.ti.com*>

Texas Instruments

XS code added by Greg Bacon <*gbacon@cs.uah.edu*>

**NAME**

Sys::Syslog, openlog, closelog, setlogmask, syslog – Perl interface to the UNIX syslog(3) calls

**SYNOPSIS**

```
use Sys::Syslog; # all except setlogsock, or:
use Sys::Syslog qw(:DEFAULT setlogsock); # default set, plus setlogsock

setlogsock $sock_type;
openlog $ident, $logopt, $facility;
syslog $priority, $format, @args;
$oldmask = setlogmask $mask_priority;
closelog;
```

**DESCRIPTION**

Sys::Syslog is an interface to the UNIX syslog(3) program. Call syslog() with a string priority and a list of printf() args just like syslog(3).

Syslog provides the functions:

**openlog** \$ident, \$logopt, \$facility

*\$ident* is prepended to every message. *\$logopt* contains zero or more of the words *pid*, *ndelay*, *nowait*. The *cons* option is ignored, since the failover mechanism will drop down to the console automatically if all other media fail. *\$facility* specifies the part of the system

**syslog** \$priority, \$format, @args

If *\$priority* permits, logs (*\$format*, @args) printed as by printf(3V), with the addition that *%m* is replaced with "\$!" (the latest error message).

**setlogmask** \$mask\_priority

Sets log mask *\$mask\_priority* and returns the old mask.

**setlogsock** \$sock\_type [\$stream\_location] (added in 5.004\_02)

Sets the socket type to be used for the next call to openlog() or syslog() and returns TRUE on success, undef on failure.

A value of 'unix' will connect to the UNIX domain socket returned by the `_PATH_LOG` macro (if your system defines it) in *syslog.ph*. A value of 'stream' will connect to the stream indicated by the pathname provided as the optional second parameter. A value of 'inet' will connect to an INET socket (either tcp or udp, tried in that order) returned by `getservbyname()`. 'tcp' and 'udp' can also be given as values. The value 'console' will send messages directly to the console, as for the 'cons' option in the logopts in openlog().

A reference to an array can also be passed as the first parameter. When this calling method is used, the array should contain a list of sock\_types which are attempted in order.

The default is to try tcp, udp, unix, stream, console.

Giving an invalid value for sock\_type will croak.

**closelog**

Closes the log file.

Note that openlog now takes three arguments, just like openlog(3).

**EXAMPLES**

```
openlog($program, 'cons,pid', 'user');
syslog('info', 'this is another test');
syslog('mail|warning', 'this is a better test: %d', time);
closelog();
```

```
syslog('debug', 'this is the last test');
setlogsock('unix');
openlog("$program $$", 'ndelay', 'user');
syslog('notice', 'fooprogram: this is really done');

setlogsock('inet');
$! = 55;
syslog('info', 'problem was %m'); # %m == $! in syslog(3)
```

**SEE ALSO**

[syslog\(3\)](#)

**AUTHOR**

Tom Christiansen <[tchrist@perl.com](mailto:tchrist@perl.com)> and Larry Wall <[larry@wall.org](mailto:larry@wall.org)>.

UNIX domain sockets added by Sean Robinson <[robinson\\_s@sc.maricopa.edu](mailto:robinson_s@sc.maricopa.edu)> with support from Tim Bunce <[Tim.Bunce@ig.co.uk](mailto:Tim.Bunce@ig.co.uk)> and the perl5-porters mailing list.

Dependency on *syslog.ph* replaced with XS code by Tom Hughes <[tom@compton.nu](mailto:tom@compton.nu)>.

Code for constant ( ) s regenerated by Nicholas Clark <[nick@ccl4.org](mailto:nick@ccl4.org)>.

Failover to different communication modes by Nick Williams <[Nick.Williams@morganstanley.com](mailto:Nick.Williams@morganstanley.com)>.

## NAME

Term::ANSIColor – Color screen output using ANSI escape sequences

## SYNOPSIS

```
use Term::ANSIColor;
print color 'bold blue';
print "This text is bold blue.\n";
print color 'reset';
print "This text is normal.\n";
print colored ("Yellow on magenta.\n", 'yellow on_magenta');
print "This text is normal.\n";
print colored ['yellow on_magenta'], "Yellow on magenta.\n";

use Term::ANSIColor qw(uncolor);
print uncolor '01;31', "\n";

use Term::ANSIColor qw(:constants);
print BOLD, BLUE, "This text is in bold blue.\n", RESET;

use Term::ANSIColor qw(:constants);
$Term::ANSIColor::AUTORESET = 1;
print BOLD BLUE "This text is in bold blue.\n";
print "This text is normal.\n";
```

## DESCRIPTION

This module has two interfaces, one through `color()` and `colored()` and the other through `constants`. It also offers the utility function `uncolor()`, which has to be explicitly imported to be used (see [SYNOPSIS](#)).

`color()` takes any number of strings as arguments and considers them to be space-separated lists of attributes. It then forms and returns the escape sequence to set those attributes. It doesn't print it out, just returns it, so you'll have to print it yourself if you want to (this is so that you can save it as a string, pass it to something else, send it to a file handle, or do anything else with it that you might care to).

`uncolor()` performs the opposite translation, turning escape sequences into a list of strings.

The recognized attributes (all of which should be fairly intuitive) are clear, reset, dark, bold, underline, underscore, blink, reverse, concealed, black, red, green, yellow, blue, magenta, on\_black, on\_red, on\_green, on\_yellow, on\_blue, on\_magenta, on\_cyan, and on\_white. Case is not significant. Underline and underscore are equivalent, as are clear and reset, so use whichever is the most intuitive to you. The color alone sets the foreground color, and on\_color sets the background color.

Note that not all attributes are supported by all terminal types, and some terminals may not support any of these sequences. Dark, blink, and concealed in particular are frequently not implemented.

Attributes, once set, last until they are unset (by sending the attribute "reset"). Be careful to do this, or otherwise your attribute will last after your script is done running, and people get very annoyed at having their prompt and typing changed to weird colors.

As an aid to help with this, `colored()` takes a scalar as the first argument and any number of attribute strings as the second argument and returns the scalar wrapped in escape codes so that the attributes will be set as requested before the string and reset to normal after the string. Alternately, you can pass a reference to an array as the first argument, and then the contents of that array will be taken as attributes and color codes and the remainder of the arguments as text to colorize.

Normally, `colored()` just puts attribute codes at the beginning and end of the string, but if you set `$Term::ANSIColor::EACHLINE` to some string, that string will be considered the line delimiter and the attribute will be set at the beginning of each line of the passed string and reset at the end of each line. This is often desirable if the output is being sent to a program like a pager that can be confused by attributes



that span lines. Normally you'll want to set `$Term::ANSIColor::EACHLINE` to `"\n"` to use this feature.

Alternately, if you import `:constants`, you can use the constants `CLEAR`, `RESET`, `BOLD`, `DARK`, `UNDERLINE`, `UNDERSCORE`, `BLINK`, `REVERSE`, `CONCEALED`, `BLACK`, `RED`, `GREEN`, `YELLOW`, `BLUE`, `MAGENTA`, `ON_BLACK`, `ON_RED`, `ON_GREEN`, `ON_YELLOW`, `ON_BLUE`, `ON_MAGENTA`, `ON_CYAN`, and `ON_WHITE` directly. These are the same as `color('attribute')` and can be used if you prefer typing:

```
print BOLD BLUE ON_WHITE "Text\n", RESET;

to

print colored ("Text\n", 'bold blue on_white');
```

When using the constants, if you don't want to have to remember to add the `, RESET` at the end of each print line, you can set `$Term::ANSIColor::AUTORESET` to a true value. Then, the display mode will automatically be reset if there is no comma after the constant. In other words, with that variable set:

```
print BOLD BLUE "Text\n";

will reset the display mode afterwards, whereas:

print BOLD, BLUE, "Text\n";
```

will not.

The subroutine interface has the advantage over the constants interface in that only two subroutines are exported into your namespace, versus twenty-two in the constants interface. On the flip side, the constants interface has the advantage of better compile time error checking, since misspelled names of colors or attributes in calls to `color()` and `colored()` won't be caught until runtime whereas misspelled names of constants will be caught at compile time. So, pollute your namespace with almost two dozen subroutines that you may not even use that often, or risk a silly bug by mistyping an attribute. Your choice, TMTOWTDI after all.

## DIAGNOSTICS

### Bad escape sequence %s

(F) You passed an invalid ANSI escape sequence to `uncolor()`.

### Bareword "%s" not allowed while "strict subs" in use

(F) You probably mistyped a constant color name such as:

```
$Foobar = FOOBAR . "This line should be blue\n";

or:
```

```
@Foobar = FOOBAR, "This line should be blue\n";
```

This will only show up under `use strict` (another good reason to run under `use strict`).

### Invalid attribute name %s

(F) You passed an invalid attribute name to either `color()` or `colored()`.

### Name "%s" used only once: possible typo

(W) You probably mistyped a constant color name such as:

```
print FOOBAR "This text is color FOOBAR\n";
```

It's probably better to always use commas after constant names in order to force the next error.

### No comma allowed after filehandle

(F) You probably mistyped a constant color name such as:

```
print FOOBAR, "This text is color FOOBAR\n";
```

Generating this fatal compile error is one of the main advantages of using the constants interface, since you'll immediately know if you mistype a color name.

#### No name for escape sequence %s

(F) The ANSI escape sequence passed to `uncolor()` contains escapes which aren't recognized and can't be translated to names.

## ENVIRONMENT

### ANSI\_COLORS\_DISABLED

If this environment variable is set, all of the functions defined by this module (`color()`, `colored()`, and all of the constants not previously used in the program) will not output any escape sequences and instead will just return the empty string or pass through the original text as appropriate. This is intended to support easy use of scripts using this module on platforms that don't support ANSI escape sequences.

For it to have its proper effect, this environment variable must be set before any color constants are used in the program.

## RESTRICTIONS

It would be nice if one could leave off the commas around the constants entirely and just say:

```
print BOLD BLUE ON_WHITE "Text\n" RESET;
```

but the syntax of Perl doesn't allow this. You need a comma after the string. (Of course, you may consider it a bug that commas between all the constants aren't required, in which case you may feel free to insert commas unless you're using `$Term::ANSIColor::AUTORESET`.)

For easier debugging, you may prefer to always use the commas when not setting `$Term::ANSIColor::AUTORESET` so that you'll get a fatal compile error rather than a warning.

## NOTES

The codes generated by this module are standard terminal control codes, complying with ECMA-48 and ISO 6429 (generally referred to as "ANSI color" for the color codes). The non-color control codes (bold, dark, italic, underline, and reverse) are part of the earlier ANSI X3.64 standard for control sequences for video terminals and peripherals.

Note that not all displays are ISO 6429-compliant, or even X3.64-compliant (or are even attempting to be so). This module will not work as expected on displays that do not honor these escape sequences, such as `cmd.exe`, `4nt.exe`, and `command.com` under either Windows NT or Windows 2000. They may just be ignored, or they may display as an ESC character followed by some apparent garbage.

Jean Delvare provided the following table of different common terminal emulators and their support for the various attributes:

|          | clear | bold    | dark | under | blink      | reverse | conceal |
|----------|-------|---------|------|-------|------------|---------|---------|
| xterm    | yes   | yes     | no   | yes   | bold       | yes     | yes     |
| linux    | yes   | yes     | yes  | bold  | yes        | yes     | no      |
| rxvt     | yes   | yes     | no   | yes   | bold/black | yes     | no      |
| dtterm   | yes   | yes     | yes  | yes   | reverse    | yes     | yes     |
| teraterm | yes   | reverse | no   | yes   | rev/red    | yes     | no      |
| aixterm  | kinda | normal  | no   | yes   | no         | yes     | yes     |

Where the entry is other than yes or no, that emulator interpret the given attribute as something else instead. Note that on an `aixterm`, `clear` doesn't reset colors; you have to explicitly set the colors back to what you want. More entries in this table are welcome.

Note that codes 3 (italic), 6 (rapid blink), and 9 (strikethrough) are specified in ANSI X3.64 and ECMA-048 but are not commonly supported by most displays and emulators and therefore aren't supported by this module at the present time. ECMA-048 also specifies a large number of other attributes, including a

sequence of attributes for font changes, Fraktur characters, double-underlining, framing, circling, and overlining. As none of these attributes are widely supported or useful, they also aren't currently supported by this module.

**SEE ALSO**

ECMA-048 is available on-line (at least at the time of this writing) at  
<http://www.ecma.ch/ecma1/STAND/ECMA-048.HTM>.

ISO 6429 is available from ISO for a charge; the author of this module does not own a copy of it. Since the source material for ISO 6429 was ECMA-048 and the latter is available for free, there seems little reason to obtain the ISO standard.

The current version of this module is always available from its web site at  
<http://www.eyrie.org/~eagle/software/ansicolor/>. It is also part of the Perl core distribution as of 5.6.0.

**AUTHORS**

Original idea (using constants) by Zenin, reimplemented using subs by Russ Allbery <rra@stanford.edu>, and then combined with the original idea by Russ with input from Zenin. Russ Allbery now maintains this module.

**COPYRIGHT AND LICENSE**

Copyright 1996, 1997, 1998, 2000, 2001, 2002 Russ Allbery <rra@stanford.edu> and Zenin <zenin@bawdycafe.org>. This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

**NAME**

Term::Cap – Perl termcap interface

**SYNOPSIS**

```
require Term::Cap;
$terminal = Tgetent Term::Cap { TERM => undef, OSPEED => $ospeed };
$terminal->Trequire(qw/ce ku kd/);
$terminal->Tgoto('cm', $col, $row, $FH);
$terminal->Tputs('dl', $count, $FH);
$terminal->Tpad($string, $count, $FH);
```

**DESCRIPTION**

These are low-level functions to extract and use capabilities from a terminal capability (termcap) database.

More information on the terminal capabilities will be found in the termcap manpage on most Unix-like systems.

**METHODS**

The output strings for **Tputs** are cached for counts of 1 for performance. **Tgoto** and **Tpad** do not cache. `$self->{_xx}` is the raw termcap data and `$self->{xx}` is the cached version.

```
print $terminal->Tpad($self->{_xx}, 1);
```

**Tgoto**, **Tputs**, and **Tpad** return the string and will also output the string to `$FH` if specified.

**Tgetent**

Returns a blessed object reference which the user can then use to send the control strings to the terminal using **Tputs** and **Tgoto**.

The function extracts the entry of the specified terminal type *TERM* (defaults to the environment variable *TERM*) from the database.

It will look in the environment for a *TERMCAP* variable. If found, and the value does not begin with a slash, and the terminal type name is the same as the environment string *TERM*, the *TERMCAP* string is used instead of reading a termcap file. If it does begin with a slash, the string is used as a path name of the termcap file to search. If *TERMCAP* does not begin with a slash and name is different from *TERM*, **Tgetent** searches the files *\$HOME/.termcap*, */etc/termcap*, and */usr/share/misc/termcap*, in that order, unless the environment variable *TERMPATH* exists, in which case it specifies a list of file pathnames (separated by spaces or colons) to be searched **instead**. Whenever multiple files are searched and a tc field occurs in the requested entry, the entry it names must be found in the same file or one of the succeeding files. If there is a `:tc=...` in the *TERMCAP* environment variable string it will continue the search in the files as above.

The extracted termcap entry is available in the object as `$self->{TERMCAP}`.

It takes a hash reference as an argument with two optional keys:

**OSPEED**

The terminal output bit rate (often mistakenly called the baud rate) for this terminal – if not set a warning will be generated and it will be defaulted to 9600. *OSPEED* can be specified as either a POSIX termios/SYSV termio speeds (where 9600 equals 9600) or an old DSD-style speed (where 13 equals 9600).

**TERM**

The terminal type whose termcap entry will be used – if not supplied it will default to `$ENV{TERM}`: if that is not set then **Tgetent** will croak.

It calls `croak` on failure.

**Tpad**

Outputs a literal string with appropriate padding for the current terminal.

It takes three arguments:

**\$string**

The literal string to be output. If it starts with a number and an optional '\*' then the padding will be increased by an amount relative to this number, if the '\*' is present then this amount will be multiplied by \$cnt. This part of \$string is removed before output/

**\$cnt**

Will be used to modify the padding applied to string as described above.

**\$FH**

An optional filehandle (or IO::Handle ) that output will be printed to.

The padded \$string is returned.

**Tputs**

Output the string for the given capability padded as appropriate without any parameter substitution.

It takes three arguments:

**\$cap**

The capability whose string is to be output.

**\$cnt**

A count passed to Tpad to modify the padding applied to the output string. If \$cnt is zero or one then the resulting string will be cached.

**\$FH**

An optional filehandle (or IO::Handle ) that output will be printed to.

The appropriate string for the capability will be returned.

**Tgoto**

**Tgoto** decodes a cursor addressing string with the given parameters.

There are four arguments:

**\$cap**

The name of the capability to be output.

**\$col**

The first value to be substituted in the output string ( usually the column in a cursor addressing capability )

**\$row**

The second value to be substituted in the output string (usually the row in cursor addressing capabilities)

**\$FH**

An optional filehandle (or IO::Handle ) to which the output string will be printed.

Substitutions are made with \$col and \$row in the output string with the following sprintf() line formats:

```
%% output '%'
%d output value as in printf %d
%2 output value as in printf %2d
%3 output value as in printf %3d
```

```

%. output value as in printf %c
%+x add x to value, then do %.

%>xy if value > x then add y, no output
%r reverse order of two parameters, no output
%i increment by one, no output
%B BCD (16*(value/10)) + (value%10), no output

%n exclusive-or all parameters with 0140 (Datamedia 2500)
%D Reverse coding (value - 2*(value%16)), no output (Delta Data)

```

The output string will be returned.

### Trequire

Takes a list of capabilities as an argument and will croak if one is not found.

### EXAMPLES

```

use Term::Cap;

Get terminal output speed
require POSIX;
my $termios = new POSIX::Termios;
$termios->getattr;
my $ospeed = $termios->getospeed;

Old-style ioctl code to get ospeed:
require 'ioctl.pl';
ioctl(TTY,$TIOCGGETP,$sgtty);
($ispeed,$ospeed) = unpack('cc',$sgtty);

allocate and initialize a terminal structure
$terminal = Tgetent Term::Cap { TERM => undef, OSPEED => $ospeed };

require certain capabilities to be available
$terminal->Trequire(qw/ce ku kd/);

Output Routines, if $FH is undefined these just return the string

Tgoto does the % expansion stuff with the given args
$terminal->Tgoto('cm', $col, $row, $FH);

Tputs doesn't do any % expansion.
$terminal->Tputs('dl', $count = 1, $FH);

```

### COPYRIGHT AND LICENSE

Please see the README file in distribution.

### AUTHOR

This module is part of the core Perl distribution and is also maintained for CPAN by Jonathan Stowe  
<jns@gellyfish.com.

### SEE ALSO

termcap(5)

**NAME**

Term::Complete – Perl word completion module

**SYNOPSIS**

```
$input = Complete('prompt_string', \@completion_list);
$input = Complete('prompt_string', @completion_list);
```

**DESCRIPTION**

This routine provides word completion on the list of words in the array (or array ref).

The tty driver is put into raw mode and restored using an operating system specific command, in UNIX-like environments `stty`.

The following command characters are defined:

<tab>

Attempts word completion. Cannot be changed.

**^D** Prints completion list. Defined by `$Term::Complete::complete`.

**^U** Erases the current input. Defined by `$Term::Complete::kill`.

<del>, <bs>

Erases one character. Defined by `$Term::Complete::erase1` and `$Term::Complete::erase2`.

**DIAGNOSTICS**

Bell sounds when word completion fails.

**BUGS**

The completion character <tab> cannot be changed.

**AUTHOR**

Wayne Thompson

**NAME**

Term::ReadLine – Perl interface to various readline packages. If no real package is found, substitutes stubs instead of basic functions.

**SYNOPSIS**

```
use Term::ReadLine;
my $term = new Term::ReadLine 'Simple Perl calc';
my $prompt = "Enter your arithmetic expression: ";
my $OUT = $term->OUT || *STDOUT;
while (defined ($_ = $term->readline($prompt))) {
 my $res = eval($_, "\n");
 warn $@ if $@;
 print $OUT $res, "\n" unless $@;
 $term->addhistory($_) if /\S/;
}
```

**DESCRIPTION**

This package is just a front end to some other packages. At the moment this description is written, the only such package is Term-ReadLine, available on CPAN near you. The real target of this stub package is to set up a common interface to whatever Readline emerges with time.

**Minimal set of supported functions**

All the supported functions should be called as methods, i.e., either as

```
$term = new Term::ReadLine 'name';
```

or as

```
$term->addhistory('row');
```

where \$term is a return value of Term::ReadLine->Init.

|             |                                                                                                                                                                                                                                                                       |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ReadLine    | returns the actual package that executes the commands. Among possible values are Term::ReadLine::Gnu, Term::ReadLine::Perl, Term::ReadLine::Stub.                                                                                                                     |
| new         | returns the handle for subsequent calls to following functions. Argument is the name of the application. Optionally can be followed by two arguments for IN and OUT filehandles. These arguments should be globs.                                                     |
| readline    | gets an input line, <i>possibly</i> with actual readline support. Trailing newline is removed. Returns undef on EOF.                                                                                                                                                  |
| addhistory  | adds the line to the history of input, from where it can be used if the actual readline is present.                                                                                                                                                                   |
| IN, \$OUT   | return the filehandles for input and output or undef if readline input and output cannot be used for Perl.                                                                                                                                                            |
| MinLine     | If argument is specified, it is an advice on minimal size of line to be included into history. undef means do not include anything into history. Returns the old value.                                                                                               |
| findConsole | returns an array with two strings that give most appropriate names for files for input and output using conventions "<\$in", ">out".                                                                                                                                  |
| Attribs     | returns a reference to a hash which describes internal configuration of the package. Names of keys in this hash conform to standard conventions with the leading rl_ stripped.                                                                                        |
| Features    | Returns a reference to a hash with keys being features present in current implementation. Several optional features are used in the minimal interface: appname should be present if the first argument to new is recognized, and minline should be present if MinLine |



method is not dummy. `autohistory` should be present if lines are put into history automatically (maybe subject to `MinLine`), and `addhistory` if `addhistory` method is not dummy.

If `Features` method reports a feature `attrs` as present, the method `Attrs` is not dummy.

### Additional supported functions

Actually `Term::ReadLine` can use some other package, that will support richer set of commands.

All these commands are callable via method interface and have names which conform to standard conventions with the leading `rl_` stripped.

The stub package included with the perl distribution allows some additional methods:

|                        |                                                                                                                                                                                                                                                                                                                                                  |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>tkRunning</code> | makes Tk event loop run when waiting for user input (i.e., during <code>readline</code> method).                                                                                                                                                                                                                                                 |
| <code>ornaments</code> | makes the command line stand out by using termcap data. The argument to <code>ornaments</code> should be 0, 1, or a string of a form <code>"aa,bb,cc,dd"</code> . Four components of this string should be names of <i>terminal capacities</i> , first two will be issued to make the prompt standout, last two to make the input line standout. |
| <code>newTTY</code>    | takes two arguments which are input filehandle and output filehandle. Switches to use these filehandles.                                                                                                                                                                                                                                         |

One can check whether the currently loaded `ReadLine` package supports these methods by checking for corresponding `Features`.

### EXPORTS

None

### ENVIRONMENT

The environment variable `PERL_RL` governs which `ReadLine` clone is loaded. If the value is false, a dummy interface is used. If the value is true, it should be tail of the name of the package to use, such as `Perl` or `Gnu`.

As a special case, if the value of this variable is space-separated, the tail might be used to disable the ornaments by setting the tail to be `o=0` or `ornaments=0`. The head should be as described above, say

If the variable is not set, or if the head of space-separated list is empty, the best available package is loaded.

```
export "PERL_RL=Perl o=0" # Use Perl ReadLine without ornaments
export "PERL_RL= o=0" # Use best available ReadLine without ornaments
```

(Note that processing of `PERL_RL` for ornaments is in the discretion of the particular used `Term::ReadLine::*` package).

### CAVEATS

It seems that using `Term::ReadLine` from Emacs minibuffer doesn't work quite right and one will get an error message like

```
Cannot open /dev/tty for read at ...
```

One possible workaround for this is to explicitly open `/dev/tty` like this

```
open (FH, "/dev/tty")
or eval 'sub Term::ReadLine::findConsole { ("&STDIN", "&STDERR") }';
die $@ if $@;
close (FH);
```

or you can try using the 4-argument form of `Term::ReadLine-new()`.

**NAME**

Test – provides a simple framework for writing test scripts

**SYNOPSIS**

```
use strict;
use Test;

use a BEGIN block so we print our plan before MyModule is loaded
BEGIN { plan tests => 14, todo => [3,4] }

load your module...
use MyModule;

ok(0); # failure
ok(1); # success

ok(0); # ok, expected failure (see todo list, above)
ok(1); # surprise success!

ok(0,1); # failure: '0' ne '1'
ok('broke','fixed'); # failure: 'broke' ne 'fixed'
ok('fixed','fixed'); # success: 'fixed' eq 'fixed'
ok('fixed',qr/x/); # success: 'fixed' =~ qr/x/

ok(sub { 1+1 }, 2); # success: '2' eq '2'
ok(sub { 1+1 }, 3); # failure: '2' ne '3'
ok(0, int(rand(2))); # (just kidding :-)

my @list = (0,0);
ok @list, 3, "@list=".join(', ', @list); #extra diagnostics
ok 'segmentation fault', '/(?i)success/'; #regex match

skip($feature_is_missing, ...); #do platform specific test
```

**DESCRIPTION**

**STOP!** If you are writing a new test, we *highly suggest* you use the new `Test::Simple` and `Test::More` modules instead.

[\*Test::Harness\*](#)/[\*Test::Harness\*](#) expects to see particular output when it executes tests. This module aims to make writing proper test scripts just a little bit easier (and less error prone :-).

**Functions**

All the following are exported by Test by default.

**plan**

```
BEGIN { plan %theplan; }
```

This should be the first thing you call in your test script. It declares your testing plan, how many there will be, if any of them should be allowed to fail, etc...

Typical usage is just:

```
use Test;
BEGIN { plan tests => 23 }
```

Things you can put in the plan:

|        |                                                                                                        |
|--------|--------------------------------------------------------------------------------------------------------|
| tests  | The number of tests in your script.<br>This means all <code>ok()</code> and <code>skip()</code> calls. |
| todo   | A reference to a list of tests which are allowed to fail. See <code>L&lt;TODO TESTS&gt;</code> .       |
| onfail | A subroutine reference to be run at the end of                                                         |

the test script should any of the tests fail.  
See `L</ONFAIL>`.

You must call `plan()` once and only once.

### `_to_value`

```
my $value = _to_value($input);
```

Converts an `ok` parameter to its value. Typically this just means running it if its a code reference. You should run all inputted values through this.

### `ok`

```
ok(1 + 1 == 2);
ok($have, $expect);
ok($have, $expect, $diagnostics);
```

This is the reason for `Test`'s existence. Its the basic function that handles printing "ok" or "not ok" along with the current test number.

In its most basic usage, it simply takes an expression. If its true, the test passes, if false, the test fails. Simp.

```
ok(1 + 1 == 2); # ok if 1 + 1 == 2
ok($foo =~ /bar/); # ok if $foo contains 'bar'
ok(baz($x + $y) eq 'Armondo'); # ok if baz($x + $y) returns
 # 'Armondo'
ok(@a == @b); # ok if @a and @b are the same length
```

The expression is evaluated in scalar context. So the following will work:

```
ok(@stuff); # ok if @stuff has any elements
ok(!grep !defined $_, @stuff); # ok if everything in @stuff is
 # defined.
```

A special case is if the expression is a subroutine reference. In that case, it is executed and its value (true or false) determines if the test passes or fails.

In its two argument form it compares the two values to see if they equal (with `eq`).

```
ok("this", "that"); # not ok, 'this' ne 'that'
```

If either is a subroutine reference, that is run and used as a comparison.

Should `$expect` either be a regex reference (ie. `qr/`) or a string that looks like a regex (ie. `'/foo/'`) `ok()` will perform a pattern match against it rather than using `eq`.

```
ok('JaffO', '/Jaff/'); # ok, 'JaffO' =~ /Jaff/
ok('JaffO', qr/Jaff/); # ok, 'JaffO' =~ qr/Jaff/;
ok('JaffO', '/(?i)jaff/'); # ok, 'JaffO' =~ /jaff/i;
```

Finally, an optional set of `$diagnostics` will be printed should the test fail. This should usually be some useful information about the test pertaining to why it failed or perhaps a description of the test. Or both.

```
ok(grep($_ eq 'something unique', @stuff), 1,
 "Something that should be unique isn't!\n".
 '@stuff = ' . join ' ', @stuff
);
```

Unfortunately, a diagnostic cannot be used with the single argument style of `ok()`.

All these special cases can cause some problems. See [/BUGS and CAVEATS](#).

## TEST TYPES

### • NORMAL TESTS

These tests are expected to succeed. If they don't something's screwed up!

### • SKIPPED TESTS

Skip is for tests that might or might not be possible to run depending on the availability of platform specific features. The first argument should evaluate to true (think "yes, please skip") if the required feature is not available. After the first argument, skip works exactly the same way as do normal tests.

### • TODO TESTS

TODO tests are designed for maintaining an **executable TODO list**. These tests are expected NOT to succeed. If a TODO test does succeed, the feature in question should not be on the TODO list, now should it?

Packages should NOT be released with succeeding TODO tests. As soon as a TODO test starts working, it should be promoted to a normal test and the newly working feature should be documented in the release notes or change log.

## ONFAIL

```
BEGIN { plan test => 4, onfail => sub { warn "CALL 911!" } }
```

While test failures should be enough, extra diagnostics can be triggered at the end of a test run. `onfail` is passed an array ref of hash refs that describe each test failure. Each hash will contain at least the following fields: `package`, `repetition`, and `result`. (The file, line, and test number are not included because their correspondence to a particular test is tenuous.) If the test had an expected value or a diagnostic string, these will also be included.

The **optional** `onfail` hook might be used simply to print out the version of your package and/or how to report problems. It might also be used to generate extremely sophisticated diagnostics for a particularly bizarre test failure. However it's not a panacea. Core dumps or other unrecoverable errors prevent the `onfail` hook from running. (It is run inside an `END` block.) Besides, `onfail` is probably over-kill in most cases. (Your test code should be simpler than the code it is testing, yes?)

## BUGS and CAVEATS

`ok()`'s special handling of subroutine references is an unfortunate "feature" that can't be removed due to compatibility.

`ok()`'s use of string eq can sometimes cause odd problems when comparing numbers, especially if you're casting a string to a number:

```
$foo = "1.0";
ok($foo, 1); # not ok, "1.0" ne 1
```

Your best bet is to use the single argument form:

```
ok($foo == 1); # ok "1.0" == 1
```

`ok()`'s special handling of strings which look like they might be regexes can also cause unexpected behavior. An innocent:

```
ok($fileglob, '/path/to/some/*stuff/');
```

will fail since `Test.pm` considers the second argument to a regex. Again, best bet is to use the single argument form:

```
ok($fileglob eq '/path/to/some/*stuff/');
```

## NOTE

This module is no longer actively being developed, only bug fixes and small tweaks (I'll still accept patches). If you desire additional functionality, consider [Test::More](#) or [Test::Unit](#).

**SEE ALSO**

*Test::Simple*, *Test::More*, *Test::Harness*, *Devel::Cover*

*Test::Builder* for building your own testing library.

*Test::Unit* is an interesting XUnit-style testing library.

*Test::Inline* and *SelfTest* let you embed tests in code.

**AUTHOR**

Copyright (c) 1998–2000 Joshua Nathaniel Pritikin. All rights reserved. Copyright (c) 2001–2002 Michael G Schwern.

Current maintainer, Michael G Schwern <schwern@pobox.com>

This package is free software and is provided "as is" without express or implied warranty. It may be used, redistributed and/or modified under the same terms as Perl itself.

## NAME

Test::Builder – Backend for building test libraries

## SYNOPSIS

```
package My::Test::Module;
use Test::Builder;
require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(ok);

my $Test = Test::Builder->new;
$Test->output('my_logfile');

sub import {
 my($self) = shift;
 my $pack = caller;

 $Test->exported_to($pack);
 $Test->plan(@_);

 $self->export_to_level(1, $self, 'ok');
}

sub ok {
 my($test, $name) = @_;
 $Test->ok($test, $name);
}
```

## DESCRIPTION

Test::Simple and Test::More have proven to be popular testing modules, but they're not always flexible enough. Test::Builder provides the a building block upon which to write your own test libraries *which can work together*.

## Construction

### new

```
my $Test = Test::Builder->new;
```

Returns a Test::Builder object representing the current state of the test.

Since you only run one test per program, there is **one and only one** Test::Builder object. No matter how many times you call `new()`, you're getting the same object. (This is called a singleton).

## Setting up tests

These methods are for setting up tests and declaring how many there are. You usually only want to call one of these methods.

### exported\_to

```
my $pack = $Test->exported_to;
$Test->exported_to($pack);
```

Tells Test::Builder what package you exported your functions to. This is important for getting TODO tests right.

### plan

```
$Test->plan('no_plan');
$Test->plan(skip_all => $reason);
$Test->plan(tests => $num_tests);
```

A convenient way to set up your tests. Call this and Test::Builder will print the appropriate headers

and take the appropriate actions.

If you call `plan()`, don't call any of the other methods below.

### **expected\_tests**

```
my $max = $Test->expected_tests;
$Test->expected_tests($max);
```

Gets/sets the # of tests we expect this test to run and prints out the appropriate headers.

### **no\_plan**

```
$Test->no_plan;
```

Declares that this test will run an indeterminate # of tests.

### **skip\_all**

```
$Test->skip_all;
$Test->skip_all($reason);
```

Skips all the tests, using the given `$reason`. Exits immediately with 0.

## **Running tests**

These actually run the tests, analogous to the functions in `Test::More`.

`$name` is always optional.

### **ok**

```
$Test->ok($test, $name);
```

Your basic test. Pass if `$test` is true, fail if `$test` is false. Just like `Test::Simple`'s `ok()`.

### **is\_eq**

```
$Test->is_eq($got, $expected, $name);
```

Like `Test::More`'s `is()`. Checks if `$got` eq `$expected`. This is the string version.

### **is\_num**

```
$Test->is_num($got, $expected, $name);
```

Like `Test::More`'s `is()`. Checks if `$got` == `$expected`. This is the numeric version.

### **isnt\_eq**

```
$Test->isnt_eq($got, $dont_expect, $name);
```

Like `Test::More`'s `isnt()`. Checks if `$got` ne `$dont_expect`. This is the string version.

### **isnt\_num**

```
$Test->isnt_num($got, $dont_expect, $name);
```

Like `Test::More`'s `isnt()`. Checks if `$got` ne `$dont_expect`. This is the numeric version.

### **like**

```
$Test->like($this, qr/$regex/, $name);
$Test->like($this, '/$regex/', $name);
```

Like `Test::More`'s `like()`. Checks if `$this` matches the given `$regex`.

You'll want to avoid `qr//` if you want your tests to work before 5.005.

### **unlike**

```
$Test->unlike($this, qr/$regex/, $name);
$Test->unlike($this, '/$regex/', $name);
```

Like `Test::More`'s `unlike()`. Checks if `$this` **does not match** the given `$regex`.

**maybe\_regex**

```
$Test->maybe_regex(qr/$regex/);
$Test->maybe_regex('/$regex/');
```

Convenience method for building testing functions that take regular expressions as arguments, but need to work before perl 5.005.

Takes a quoted regular expression produced by qr//, or a string representing a regular expression.

Returns a Perl value which may be used instead of the corresponding regular expression, or undef if it's argument is not recognised.

For example, a version of like(), sans the useful diagnostic messages, could be written as:

```
sub laconic_like {
 my ($self, $this, $regex, $name) = @_;
 my $usable_regex = $self->maybe_regex($regex);
 die "expecting regex, found '$regex'\n"
 unless $usable_regex;
 $self->ok($this =~ m/$usable_regex/, $name);
}
```

**cmp\_ok**

```
$Test->cmp_ok($this, $type, $that, $name);
```

Works just like Test::More's cmp\_ok().

```
$Test->cmp_ok($big_num, '!=', $other_big_num);
```

**BAILOUT**

```
$Test->BAILOUT($reason);
```

Indicates to the Test::Harness that things are going so badly all testing should terminate. This includes running any additional test scripts.

It will exit with 255.

**skip**

```
$Test->skip;
$Test->skip($why);
```

Skips the current test, reporting \$why.

**todo\_skip**

```
$Test->todo_skip;
$Test->todo_skip($why);
```

Like skip(), only it will declare the test as failing and TODO. Similar to

```
print "not ok $tnum # TODO $why\n";
```

**skip\_rest**

```
$Test->skip_rest;
$Test->skip_rest($reason);
```

Like skip(), only it skips all the rest of the tests you plan to run and terminates the test.

If you're running under no\_plan, it skips once and terminates the test.

=end\_unimplemented



## Test style

### level

```
$Test->level($show_high);
```

How far up the call stack should \$Test look when reporting where the test failed.

Defaults to 1.

Setting \$Test::Builder::Level overrides. This is typically useful localized:

```
{
 local $Test::Builder::Level = 2;
 $Test->ok($test);
}
```

### use\_numbers

```
$Test->use_numbers($on_or_off);
```

Whether or not the test should output numbers. That is, this if true:

```
ok 1
ok 2
ok 3
```

or this if false

```
ok
ok
ok
```

Most useful when you can't depend on the test output order, such as when threads or forking is involved.

Test::Harness will accept either, but avoid mixing the two styles.

Defaults to on.

### no\_header

```
$Test->no_header($no_header);
```

If set to true, no "1..N" header will be printed.

### no\_ending

```
$Test->no_ending($no_ending);
```

Normally, Test::Builder does some extra diagnostics when the test ends. It also changes the exit code as described in Test::Simple.

If this is true, none of that will be done.

## Output

Controlling where the test output goes.

It's ok for your test to change where STDOUT and STDERR point to, Test::Builder's default output settings will not be affected.

### diag

```
$Test->diag(@msgs);
```

Prints out the given \$message. Normally, it uses the failure\_output() handle, but if this is for a TODO test, the todo\_output() handle is used.

Output will be indented and marked with a # so as not to interfere with test output. A newline will be

put on the end if there isn't one already.

We encourage using this rather than calling `print` directly.

Returns false. Why? Because `diag()` is often used in conjunction with a failing test (`ok()` || `diag()`) it "passes through" the failure.

```
return ok(...) || diag(...);
```

=for blame transfer Mark Fowler <mark@twoshortplanks.com>

### **\_print**

```
$Test->_print(@msgs);
```

Prints to the `output()` filehandle.

=end\_private

### **output**

```
$Test->output($fh);
$Test->output($file);
```

Where normal "ok/not ok" test output should go.

Defaults to `STDOUT`.

### **failure\_output**

```
$Test->failure_output($fh);
$Test->failure_output($file);
```

Where diagnostic output on test failures and `diag()` should go.

Defaults to `STDERR`.

### **todo\_output**

```
$Test->todo_output($fh);
$Test->todo_output($file);
```

Where diagnostics about todo test failures and `diag()` should go.

Defaults to `STDOUT`.

## **Test Status and Info**

### **current\_test**

```
my $curr_test = $Test->current_test;
$Test->current_test($num);
```

Gets/sets the current test # we're on.

You usually shouldn't have to set this.

### **summary**

```
my @tests = $Test->summary;
```

A simple summary of the tests so far. True for pass, false for fail. This is a logical pass/fail, so todos are passes.

Of course, test #1 is `$tests[0]`, etc...

### **details** *UNIMPLEMENTED*

```
my @tests = $Test->details;
```

Like `summary()`, but with a lot more detail.

```
$tests[$test_num - 1] =
```

```

{ ok => is the test considered ok?
 actual_ok => did it literally say 'ok'?
 name => name of the test (if any)
 type => 'skip' or 'todo' (if any)
 reason => reason for the above (if any)
};

```

**todo**

```

my $todo_reason = $Test->todo;
my $todo_reason = $Test->todo($pack);

```

`todo()` looks for a `$TODO` variable in your tests. If set, all tests will be considered 'todo' (see `Test::More` and `Test::Harness` for details). Returns the reason (ie. the value of `$TODO`) if running as todo tests, false otherwise.

`todo()` is pretty part about finding the right package to look for `$TODO` in. It uses the `exported_to()` package to find it. If that's not set, it's pretty good at guessing the right package to look at.

Sometimes there is some confusion about where `todo()` should be looking for the `$TODO` variable. If you want to be sure, tell it explicitly what `$pack` to use.

**caller**

```

my $package = $Test->caller;
my($pack, $file, $line) = $Test->caller;
my($pack, $file, $line) = $Test->caller($height);

```

Like the normal `caller()`, except it reports according to your `level()`.

**\_sanity\_check**

```
_sanity_check();
```

Runs a bunch of end of test sanity checks to make sure reality came through ok. If anything is wrong it will die with a fairly friendly error message.

**\_whoa**

```
_whoa($check, $description);
```

A sanity check, similar to `assert()`. If the `$check` is true, something has gone horribly wrong. It will die with the given `$description` and a note to contact the author.

**\_my\_exit**

```
_my_exit($exit_num);
```

Perl seems to have some trouble with exiting inside an `END` block. 5.005\_03 and 5.6.1 both seem to do odd things. Instead, this function edits `$?` directly. It should ONLY be called from inside an `END` block. It doesn't actually exit, that's your job.

```
=end_private
```

**THREADS**

In perl 5.8.0 and later, `Test::Builder` is thread-safe. The test number is shared amongst all threads. This means if one thread sets the test number using `current_test()` they will all be effected.

**EXAMPLES**

CPAN can provide the best examples. `Test::Simple`, `Test::More`, `Test::Exception` and `Test::Differences` all use `Test::Builder`.

**SEE ALSO**

Test::Simple, Test::More, Test::Harness

**AUTHORS**

Original code by chromatic, maintained by Michael G Schwern <schwern@pobox.com>

**COPYRIGHT**

Copyright 2001 by chromatic <chromatic@wgz.org>,  
Michael G Schwern <schwern@pobox.com>.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See *<http://www.perl.com/perl/misc/Artistic.html>*

**NAME**

Test::Harness – run perl standard test scripts with statistics

**SYNOPSIS**

```
use Test::Harness;

runtests(@test_files);
```

**DESCRIPTION**

**STOP!** If all you want to do is write a test script, consider using Test::Simple. Otherwise, read on.

(By using the Test module, you can write test scripts without knowing the exact output this module expects. However, if you need to know the specifics, read on!)

Perl test scripts print to standard output "ok N" for each single test, where N is an increasing sequence of integers. The first line output by a standard test script is "1..M" with M being the number of tests that should be run within the test script. Test::Harness::runtests(@tests) runs all the testscripts named as arguments and checks standard output for the expected "ok N" strings.

After all tests have been performed, runtests() prints some performance statistics that are computed by the Benchmark module.

**The test script output**

The following explains how Test::Harness interprets the output of your test program.

**‘1..M’**

This header tells how many tests there will be. For example, 1..10 means you plan on running 10 tests. This is a safeguard in case your test dies quietly in the middle of its run.

It should be the first non-comment line output by your test program.

In certain instances, you may not know how many tests you will ultimately be running. In this case, it is permitted for the 1..M header to appear as the **last** line output by your test (again, it can be followed by further comments).

Under **no** circumstances should 1..M appear in the middle of your output or more than once.

**‘ok’, ‘not ok’. Ok?**

Any output from the testscript to standard error is ignored and bypassed, thus will be seen by the user. Lines written to standard output containing `/^(not\s+)?ok\b/` are interpreted as feedback for runtests(). All other lines are discarded.

`/^not ok/` indicates a failed test. `/^ok/` is a successful test.

**test numbers**

Perl normally expects the ‘ok’ or ‘not ok’ to be followed by a test number. It is tolerated if the test numbers after ‘ok’ are omitted. In this case Test::Harness maintains temporarily its own counter until the script supplies test numbers again. So the following test script

```
print <<END;
1..6
not ok
ok
not ok
ok
ok
END
```

will generate

```
FAILED tests 1, 3, 6
```

Failed 3/6 tests, 50.00% okay

### test names

Anything after the test number but before the # is considered to be the name of the test.

```
ok 42 this is the name of the test
```

Currently, Test::Harness does nothing with this information.

### Skipping tests

If the standard output line contains the substring `# Skip` (with variations in spacing and case) after `ok` or `ok NUMBER`, it is counted as a skipped test. If the whole testscript succeeds, the count of skipped tests is included in the generated output. Test::Harness reports the text after `# Skip\S*\s+` as a reason for skipping.

```
ok 23 # skip Insufficient flogiston pressure.
```

Similarly, one can include a similar explanation in a `1..0` line emitted if the test script is skipped completely:

```
1..0 # Skipped: no leverage found
```

### Todo tests

If the standard output line contains the substring `# TODO` after `not ok` or `not ok NUMBER`, it is counted as a todo test. The text afterwards is the thing that has to be done before this test will succeed.

```
not ok 13 # TODO harness the power of the atom
=end _deprecated
```

Alternatively, you can specify a list of what tests are todo as part of the test header.

```
1..23 todo 5 12 23
```

This only works if the header appears at the beginning of the test.

This style is **deprecated**.

```
=end _deprecated
```

These tests represent a feature to be implemented or a bug to be fixed and act as something of an executable "thing to do" list. They are **not** expected to succeed. Should a todo test begin succeeding, Test::Harness will report it as a bonus. This indicates that whatever you were supposed to do has been done and you should promote this to a normal test.

### Bail out!

As an emergency measure, a test script can decide that further tests are useless (e.g. missing dependencies) and testing should stop immediately. In that case the test script prints the magic words

```
Bail out!
```

to standard output. Any message after these words will be displayed by Test::Harness as the reason why testing is stopped.

### Comments

Additional comments may be put into the testing output on their own lines. Comment lines should begin with a '#', Test::Harness will ignore them.

```
ok 1
Life is good, the sun is shining, RAM is cheap.
not ok 2
got 'Bush' expected 'Gore'
```

**Anything else**

Any other output Test::Harness sees it will silently ignore **BUT WE PLAN TO CHANGE THIS!** If you wish to place additional output in your test script, please use a comment.

**Taint mode**

Test::Harness will honor the `-T` in the `#!` line on your test files. So if you begin a test with:

```
#!/perl -T
```

the test will be run with taint mode on.

**Configuration variables.**

These variables can be used to configure the behavior of Test::Harness. They are exported on request.

**\$Test::Harness::verbose**

The global variable `$Test::Harness::verbose` is exportable and can be used to let `runtests()` display the standard output of the script without altering the behavior otherwise.

**\$Test::Harness::switches**

The global variable `$Test::Harness::switches` is exportable and can be used to set perl command line options used for running the test script(s). The default value is `-w`.

**Failure**

It will happen, your tests will fail. After you mop up your ego, you can begin examining the summary report:

```
t/base.....ok
t/nonumbers.....ok
t/ok.....ok
t/test-harness.....ok
t/waterloo.....dubious
 Test returned status 3 (wstat 768, 0x300)
DIED. FAILED tests 1, 3, 5, 7, 9, 11, 13, 15, 17, 19
 Failed 10/20 tests, 50.00% okay
Failed Test Stat Wstat Total Fail Failed List of Failed

t/waterloo.t 3 768 20 10 50.00% 1 3 5 7 9 11 13 15 17 19
Failed 1/5 test scripts, 80.00% okay. 10/44 subtests failed, 77.27% okay.
```

Everything passed but `t/waterloo.t`. It failed 10 of 20 tests and exited with non-zero status indicating something dubious happened.

The columns in the summary report mean:

**Failed Test**

The test file which failed.

**Stat** If the test exited with non-zero, this is its exit status.

**Wstat**

The wait status of the test *umm, I need a better explanation here.*

**Total**

Total number of tests expected to run.

**Fail** Number which failed, either from "not ok" or because they never ran.

**Failed**

Percentage of the total tests which failed.

## List of Failed

A list of the tests which failed. Successive failures may be abbreviated (ie. 15–20 to indicate that tests 15, 16, 17, 18, 19 and 20 failed).

## Functions

Test::Harness currently only has one function, here it is.

### runtests

```
my $all_ok = runtests(@test_files);
```

This runs all the given @test\_files and divines whether they passed or failed based on their output to STDOUT (details above). It prints out each individual test which failed along with a summary report and a how long it all took.

It returns true if everything was ok. Otherwise it will die() with one of the messages in the DIAGNOSTICS section.

```
=for_private This is just _run_all_tests() plus _show_results()
```

### \_all\_ok

```
my $ok = _all_ok(\%tot);
```

Tells you if this test run is overall successful or not.

### \_globdir

```
my @files = _globdir $dir;
```

Returns all the files in a directory. This is shorthand for backwards compatibility on systems where glob() doesn't work right.

### \_run\_all\_tests

```
my($total, $failed) = _run_all_tests(@test_files);
```

Runs all the given @test\_files (as runtests()) but does it quietly (no report). \$total is a hash ref summary of all the tests run. Its keys and values are this:

|             |                                                     |
|-------------|-----------------------------------------------------|
| bonus       | Number of individual todo tests unexpectedly passed |
| max         | Number of individual tests ran                      |
| ok          | Number of individual tests passed                   |
| sub_skipped | Number of individual tests skipped                  |
| todo        | Number of individual todo tests                     |
| files       | Number of test files ran                            |
| good        | Number of test files passed                         |
| bad         | Number of test files failed                         |
| tests       | Number of test files originally given               |
| skipped     | Number of test files skipped                        |

If \$total->{bad} == 0 and \$total->{max} 0, you've got a successful test.

\$failed is a hash ref of all the test scripts which failed. Each key is the name of a test script, each value is another hash representing how that script failed. Its keys are these:

|         |                                         |
|---------|-----------------------------------------|
| name    | Name of the test which failed           |
| estat   | Script's exit value                     |
| wstat   | Script's wait status                    |
| max     | Number of individual tests              |
| failed  | Number which failed                     |
| percent | Percentage of tests which failed        |
| canon   | List of tests which failed (as string). |



Needless to say, `$failed` should be empty if everything passed.

**NOTE** Currently this function is still noisy. I'm working on it.

### **`_mk_leader`**

```
my($leader, $ml) = _mk_leader($test_file, $width);
```

Generates the `'t/foo.....'` `$leader` for the given `$test_file` as well as a similar version which will overwrite the current line (by use of `\r` and `such`). `$ml` may be empty if `Test::Harness` doesn't think you're on TTY.

The `$width` is the width of the "yada/blah.." string.

### **`_leader_width`**

```
my($width) = _leader_width(@test_files);
```

Calculates how wide the leader should be based on the length of the longest test name.

## **EXPORT**

`&runtests` is exported by `Test::Harness` per default.

`$verbose` and `$switches` are exported upon request.

## **DIAGNOSTICS**

```
All tests successful.\nFiles=%d, Tests=%d, %s
```

If all tests are successful some statistics about the performance are printed.

```
FAILED tests %s\n\tFailed %d/%d tests, %.2f%% okay.
```

For any single script that has failing subtests statistics like the above are printed.

```
Test returned status %d (wstat %d)
```

Scripts that return a non-zero exit status, both `$? > 8` and `$?` are printed in a message similar to the above.

```
Failed 1 test, %.2f%% okay. %s
```

```
Failed %d/%d tests, %.2f%% okay. %s
```

If not all tests were successful, the script dies with one of the above messages.

```
FAILED-Further testing stopped: %s
```

If a single subtest decides that further testing will not make sense, the script dies with this message.

## **ENVIRONMENT**

### **`HARNESS_ACTIVE`**

Harness sets this before executing the individual tests. This allows the tests to determine if they are being executed through the harness or by any other means.

### **`HARNESS_COLUMNS`**

This value will be used for the width of the terminal. If it is not set then it will default to `COLUMNS`. If this is not set, it will default to 80. Note that users of Bourne-sh based shells will need to `export COLUMNS` for this module to use that variable.

### **`HARNESS_COMPILE_TEST`**

When true it will make harness attempt to compile the test using `perlcc` before running it.

**NOTE** This currently only works when sitting in the perl source directory!

### **`HARNESS_FILELEAK_IN_DIR`**

When set to the name of a directory, harness will check after each test whether new files appeared in that directory, and report them as

```
LEAKED FILES: scr.tmp 0 my.db
```

If relative, directory name is with respect to the current directory at the moment `runtests()` was called. Putting absolute path into `HARNESS_FILELEAK_IN_DIR` may give more predictable results.

`HARNESS_IGNORE_EXITCODE`

Makes harness ignore the exit status of child processes when defined.

`HARNESS_NOTTY`

When set to a true value, forces it to behave as though `STDOUT` were not a console. You may need to set this if you don't want harness to output more frequent progress messages using carriage returns. Some consoles may not handle carriage returns properly (which results in a somewhat messy output).

`HARNESS_PERL_SWITCHES`

Its value will be prepended to the switches used to invoke perl on each test. For example, setting `HARNESS_PERL_SWITCHES` to `-W` will run all tests with all warnings enabled.

`HARNESS_VERBOSE`

If true, `Test::Harness` will output the verbose results of running its tests. Setting `$Test::Harness::verbose` will override this.

## EXAMPLE

Here's how `Test::Harness` tests itself

```
$ cd ~/src/devel/Test-Harness
$ perl -Mblib -e 'use Test::Harness qw(&runtests $verbose);
 $verbose=0; runtests @ARGV;' t/*.t
Using /home/schwern/src/devel/Test-Harness/blib
t/base.....ok
t/nonumbers.....ok
t/ok.....ok
t/test-harness.....ok
All tests successful.
Files=4, Tests=24, 2 wallclock secs (0.61 cusr + 0.41 csys = 1.02 CPU)
```

## SEE ALSO

[Test](#) and [Test::Simple](#) for writing test scripts, [Benchmark](#) for the underlying timing routines, [Devel::CoreStack](#) to generate core dumps from failed tests and [Devel::Cover](#) for test coverage analysis.

## AUTHORS

Either Tim Bunce or Andreas Koenig, we don't know. What we know for sure is, that it was inspired by Larry Wall's `TEST` script that came with perl distributions for ages. Numerous anonymous contributors exist. Andreas Koenig held the torch for many years.

Current maintainer is Michael G Schwern <schwern@pobox.com>

## TODO

Provide a way of running tests quietly (ie. no printing) for automated validation of tests. This will probably take the form of a version of `runtests()` which rather than printing its output returns raw data on the state of the tests. (Partially done in `Test::Harness::Straps`)

Fix `HARNESS_COMPILE_TEST` without breaking its core usage.

Figure a way to report test names in the failure summary.

Rework the test summary so long test names are not truncated as badly. (Partially done with new skip test styles)

Deal with VMS's "not \nok 4\n" mistake.

Add option for coverage analysis.

=for \_private Keeping whittling away at `_run_all_tests()`

=for \_private Clean up how the summary is printed. Get rid of those damned formats.

## BUGS

HARNESS\_COMPILE\_TEST currently assumes it's run from the Perl source directory.

**NAME**

Test::Harness::Assert – simple assert

**SYNOPSIS**

```
FOR INTERNAL USE ONLY ###

use Test::Harness::Assert;

assert(EXPR, $name);
```

**DESCRIPTION**

A simple assert routine since we don't have Carp::Assert handy.

**For internal use by Test::Harness ONLY!**

**Functions****assert**

```
assert(EXPR, $name);
```

If the expression is false the program aborts.

**AUTHOR**

Michael G Schwern <schwern@pobox.com>

**SEE ALSO**

*[Carp::Assert](#)*

**NAME**

Test::Harness::Iterator – Internal Test::Harness Iterator

**SYNOPSIS**

```
use Test::Harness::Iterator;
use Test::Harness::Iterator;
my $it = Test::Harness::Iterator->new(*TEST);
my $it = Test::Harness::Iterator->new(\@array);

my $line = $it->next;
```

**DESCRIPTION****FOR INTERNAL USE ONLY!**

This is a simple iterator wrapper for arrays and filehandles.

**NAME**

Test::Harness::Straps – detailed analysis of test results

**SYNOPSIS**

```
use Test::Harness::Straps;

my $strap = Test::Harness::Straps->new;

Various ways to interpret a test
my %results = $strap->analyze($name, \@test_output);
my %results = $strap->analyze_fh($name, $test_filehandle);
my %results = $strap->analyze_file($test_file);

UNIMPLEMENTED
my %total = $strap->total_results;

Altering the behavior of the strap UNIMPLEMENTED
my $verbose_output = $strap->dump_verbose();
$strap->dump_verbose_fh($output_filehandle);
```

**DESCRIPTION**

**THIS IS ALPHA SOFTWARE** in that the interface is subject to change in incompatible ways. It is otherwise stable.

Test::Harness is limited to printing out its results. This makes analysis of the test results difficult for anything but a human. To make it easier for programs to work with test results, we provide Test::Harness::Straps. Instead of printing the results, straps provide them as raw data. You can also configure how the tests are to be run.

The interface is currently incomplete. *Please* contact the author if you'd like a feature added or something change or just have comments.

**Construction****new**

```
my $strap = Test::Harness::Straps->new;
```

Initialize a new strap.

**\_init**

```
$strap->_init;
```

Initialize the internal state of a strap to make it ready for parsing.

**Analysis****analyze**

```
my %results = $strap->analyze($name, \@test_output);
```

Analyzes the output of a single test, assigning it the given \$name for use in the total report. Returns the %results of the test. See [Results](#).

@test\_output should be the raw output from the test, including newlines.

**analyze\_fh**

```
my %results = $strap->analyze_fh($name, $test_filehandle);
```

Like analyze, but it reads from the given filehandle.

**analyze\_file**

```
my %results = $strap->analyze_file($test_file);
```

Like `analyze`, but it runs the given `$test_file` and parses it's results. It will also use that name for the total report.

#### **`_switches`**

```
my $switches = $self->_switches($file);
```

Formats and returns the switches necessary to run the test.

#### **`_INC2PERL5LIB`**

```
local $ENV{PERL5LIB} = $self->_INC2PERL5LIB;
```

Takes the current value of `@INC` and turns it into something suitable for putting onto `PERL5LIB`.

#### **`_filtered_INC`**

```
my @filtered_inc = $self->_filtered_INC;
```

Shortens `@INC` by removing redundant and unnecessary entries. Necessary for OS's with limited command line lengths, like VMS.

#### **`_restore_PERL5LIB`**

```
$self->_restore_PERL5LIB;
```

This restores the original value of the `PERL5LIB` environment variable. Necessary on VMS, otherwise a no-op.

`=begin _private`

### **Parsing**

Methods for identifying what sort of line you're looking at.

#### **`_is_comment`**

```
my $is_comment = $strap->_is_comment($line, \$comment);
```

Checks if the given line is a comment. If so, it will place it into `$comment` (sans #).

#### **`_is_header`**

```
my $is_header = $strap->_is_header($line);
```

Checks if the given line is a header (1..M) line. If so, it places how many tests there will be in `$strap-{max}`, a list of which tests are todo in `$strap-{todo}` and if the whole test was skipped `$strap-{skip_all}` contains the reason.

#### **`_is_test`**

```
my $is_test = $strap->_is_test($line, \%test);
```

Checks if the `$line` is a test report (ie. 'ok/not ok'). Reports the result back in `%test` which will contain:

|                     |                                                       |
|---------------------|-------------------------------------------------------|
| <code>ok</code>     | did it succeed? This is the literal 'ok' or 'not ok'. |
| <code>name</code>   | name of the test (if any)                             |
| <code>number</code> | test number (if any)                                  |
| <code>type</code>   | 'todo' or 'skip' (if any)                             |
| <code>reason</code> | why is it todo or skip? (if any)                      |

If will also catch lone 'not' lines, note it saw them `$strap-{saw_lone_not}` and the line in `$strap-{lone_not_line}`.

#### **`_is_bail_out`**

```
my $is_bail_out = $strap->_is_bail_out($line, \$reason);
```

Checks if the line is a "Bail out!". Places the reason for bailing (if any) in `$reason`.

**\_reset\_file\_state**

```
$strap->_reset_file_state;
```

Resets things like \$strap-{max}, \$strap-{skip\_all}, etc... so its ready to parse the next file.

```
=end_private
```

**Results**

The %results returned from analyze() contain the following information:

|          |                                                                                     |
|----------|-------------------------------------------------------------------------------------|
| passing  | true if the whole test is considered a pass<br>(or skipped), false if its a failure |
| exit     | the exit code of the test run, if from a file                                       |
| wait     | the wait code of the test run, if from a file                                       |
| max      | total tests which should have been run                                              |
| seen     | total tests actually seen                                                           |
| skip_all | if the whole test was skipped, this will<br>contain the reason.                     |
| ok       | number of tests which passed<br>(including todo and skips)                          |
| todo     | number of todo tests seen                                                           |
| bonus    | number of todo tests which<br>unexpectedly passed                                   |
| skip     | number of tests skipped                                                             |

So a successful test should have max == seen == ok.

There is one final item, the details.

|         |                                                                    |
|---------|--------------------------------------------------------------------|
| details | an array ref reporting the result of<br>each test looks like this: |
|---------|--------------------------------------------------------------------|

```
$results{details}[$test_num - 1] =
 { ok => is the test considered ok?
 actual_ok => did it literally say 'ok'?
 name => name of the test (if any)
 type => 'skip' or 'todo' (if any)
 reason => reason for the above (if any)
 };
```

Element 0 of the details is test #1. I tried it with element 1 being #1 and 0 being empty, this is less awkward.

```
=begin_private
```

**\_detailize**

```
my %details = $strap->_detailize($pass, \%test);
```

Generates the details based on the last test line seen. \$pass is true if it was considered to be a passed test. %test is the results of the test you're summarizing.

```
=end_private
```

**EXAMPLES**

See *examples/mini\_harness.plx* for an example of use.



**AUTHOR**

Michael G Schwern <schwern@pobox.com>

**SEE ALSO**

*[Test::Harness](#)*

**NAME**

Test::More – yet another framework for writing test scripts

**SYNOPSIS**

```

use Test::More tests => $Num_Tests;
or
use Test::More qw(no_plan);
or
use Test::More skip_all => $reason;

BEGIN { use_ok('Some::Module'); }
require_ok('Some::Module');

Various ways to say "ok"
ok($this eq $that, $test_name);

is ($this, $that, $test_name);
isnt($this, $that, $test_name);

Rather than print STDERR "# here's what went wrong\n"
diag("here's what went wrong");

like ($this, qr/that/, $test_name);
unlike($this, qr/that/, $test_name);

cmp_ok($this, '==', $that, $test_name);

is_deeply($complex_structure1, $complex_structure2, $test_name);

SKIP: {
 skip $why, $show_many unless $have_some_feature;

 ok(foo(), $test_name);
 is(foo(42), 23, $test_name);
};

TODO: {
 local $TODO = $why;

 ok(foo(), $test_name);
 is(foo(42), 23, $test_name);
};

can_ok($module, @methods);
isa_ok($object, $class);

pass($test_name);
fail($test_name);

Utility comparison functions.
eq_array(\@this, \@that);
eq_hash(\%this, \%that);
eq_set(\@this, \@that);

UNIMPLEMENTED!!!
my @status = Test::More::status;

UNIMPLEMENTED!!!
BAIL_OUT($why);

```

**DESCRIPTION**

**STOP!** If you're just getting started writing tests, have a look at `Test::Simple` first. This is a drop in replacement for `Test::Simple` which you can switch to once you get the hang of basic testing.

The purpose of this module is to provide a wide range of testing utilities. Various ways to say "ok" with better diagnostics, facilities to skip tests, test future features and compare complicated data structures. While you can do almost anything with a simple `ok()` function, it doesn't provide good diagnostic output.

**I love it when a plan comes together**

Before anything else, you need a testing plan. This basically declares how many tests your script is going to run to protect against premature failure.

The preferred way to do this is to declare a plan when you use `Test::More`.

```
use Test::More tests => $Num_Tests;
```

There are rare cases when you will not know beforehand how many tests your script is going to run. In this case, you can declare that you have no plan. (Try to avoid using this as it weakens your test.)

```
use Test::More qw(no_plan);
```

In some cases, you'll want to completely skip an entire testing script.

```
use Test::More skip_all => $skip_reason;
```

Your script will declare a skip with the reason why you skipped and exit immediately with a zero (success). See [Test::Harness](#) for details.

If you want to control what functions `Test::More` will export, you have to use the 'import' option. For example, to import everything but 'fail', you'd do:

```
use Test::More tests => 23, import => ['!fail'];
```

Alternatively, you can use the `plan()` function. Useful for when you have to calculate the number of tests.

```
use Test::More;
plan tests => keys %Stuff * 3;
```

or for deciding between running the tests at all:

```
use Test::More;
if($^O eq 'MacOS') {
 plan skip_all => 'Test irrelevant on MacOS';
}
else {
 plan tests => 42;
}
```

**Test names**

By convention, each test is assigned a number in order. This is largely done automatically for you. However, it's often very useful to assign a name to each test. Which would you rather see:

```
ok 4
not ok 5
ok 6
```

or

```
ok 4 - basic multi-variable
not ok 5 - simple exponential
ok 6 - force == mass * acceleration
```

The later gives you some idea of what failed. It also makes it easier to find the test in your script, simply

search for "simple exponential".

All test functions take a name argument. It's optional, but highly suggested that you use it.

### I'm ok, you're not ok.

The basic purpose of this module is to print out either "ok #" or "not ok #" depending on if a given test succeeded or failed. Everything else is just gravy.

All of the following print "ok" or "not ok" depending on if the test succeeded or failed. They all also return true or false, respectively.

#### ok

```
ok($this eq $that, $test_name);
```

This simply evaluates any expression (`$this eq $that` is just a simple example) and uses that to determine if the test succeeded or failed. A true expression passes, a false one fails. Very simple.

For example:

```
ok($exp{9} == 81, 'simple exponential');
ok(Film->can('db_Main'), 'set_db()');
ok($p->tests == 4, 'saw tests');
ok(!grep !defined $_, @items, 'items populated');
```

(Mnemonic: "This is ok.")

`$test_name` is a very short description of the test that will be printed out. It makes it very easy to find a test in your script when it fails and gives others an idea of your intentions. `$test_name` is optional, but we **very** strongly encourage its use.

Should an `ok()` fail, it will produce some diagnostics:

```
not ok 18 - sufficient mucus
Failed test 18 (foo.t at line 42)
```

This is actually `Test::Simple`'s `ok()` routine.

#### is

#### isnt

```
is ($this, $that, $test_name);
isnt($this, $that, $test_name);
```

Similar to `ok()`, `is()` and `isnt()` compare their two arguments with `eq` and `ne` respectively and use the result of that to determine if the test succeeded or failed. So these:

```
Is the ultimate answer 42?
is(ultimate_answer(), 42, "Meaning of Life");

$foo isn't empty
isnt($foo, '', "Got some foo");
```

are similar to these:

```
ok(ultimate_answer() eq 42, "Meaning of Life");
ok($foo ne '', "Got some foo");
```

(Mnemonic: "This is that." "This isn't that.")

So why use these? They produce better diagnostics on failure. `ok()` cannot know what you are testing for (beyond the name), but `is()` and `isnt()` know what the test was and why it failed. For example this test:

```
my $foo = 'waffle'; my $bar = 'yarblokos';
is($foo, $bar, 'Is foo the same as bar?');
```

Will produce something like this:

```
not ok 17 - Is foo the same as bar?
Failed test 1 (foo.t at line 139)
got: 'waffle'
expected: 'yarblokos'
```

So you can figure out what went wrong without rerunning the test.

You are encouraged to use `is()` and `isnt()` over `ok()` where possible, however do not be tempted to use them to find out if something is true or false!

```
XXX BAD! $pope->isa('Catholic') eq 1
is($pope->isa('Catholic'), 1, 'Is the Pope Catholic?');
```

This does not check if `$pope->isa('Catholic')` is true, it checks if it returns 1. Very different. Similar caveats exist for false and 0. In these cases, use `ok()`.

```
ok($pope->isa('Catholic')), 'Is the Pope Catholic?');
```

For those grammatical pedants out there, there's an `isn't()` function which is an alias of `isnt()`.

### like

```
like($this, qr/that/, $test_name);
```

Similar to `ok()`, `like()` matches `$this` against the regex `qr/that/`.

So this:

```
like($this, qr/that/, 'this is like that');
```

is similar to:

```
ok($this =~ /that/, 'this is like that');
```

(Mnemonic "This is like that".)

The second argument is a regular expression. It may be given as a regex reference (i.e. `qr//`) or (for better compatibility with older perls) as a string that looks like a regex (alternative delimiters are currently not supported):

```
like($this, '/that/', 'this is like that');
```

Regex options may be placed on the end (`'/that/i'`).

Its advantages over `ok()` are similar to that of `is()` and `isnt()`. Better diagnostics on failure.

### unlike

```
unlike($this, qr/that/, $test_name);
```

Works exactly as `like()`, only it checks if `$this` **does not** match the given pattern.

### cmp\_ok

```
cmp_ok($this, $op, $that, $test_name);
```

Halfway between `ok()` and `is()` lies `cmp_ok()`. This allows you to compare two arguments using any binary perl operator.

```
ok($this eq $that);
cmp_ok($this, 'eq', $that, 'this eq that');

ok($this == $that);
cmp_ok($this, '==', $that, 'this == that');

ok($this && $that);
cmp_ok($this, '&&', $that, 'this || that');
```

...etc...

Its advantage over `ok()` is when the test fails you'll know what `$this` and `$that` were:

```
not ok 1
Failed test (foo.t at line 12)
'23'
&&
undef
```

It's also useful in those cases where you are comparing numbers and `is()`'s use of `eq` will interfere:

```
cmp_ok($big_hairy_number, '==', $another_big_hairy_number);
```

### **can\_ok**

```
can_ok($module, @methods);
can_ok($object, @methods);
```

Checks to make sure the `$module` or `$object` can do these `@methods` (works with functions, too).

```
can_ok('Foo', qw(this that whatever));
```

is almost exactly like saying:

```
ok(Foo->can('this') &&
 Foo->can('that') &&
 Foo->can('whatever')
);
```

only without all the typing and with a better interface. Handy for quickly testing an interface.

No matter how many `@methods` you check, a single `can_ok()` call counts as one test. If you desire otherwise, use:

```
foreach my $meth (@methods) {
 can_ok('Foo', $meth);
}
```

### **isa\_ok**

```
isa_ok($object, $class, $object_name);
isa_ok($ref, $type, $ref_name);
```

Checks to see if the given `$object` `isa($class)`. Also checks to make sure the object was defined in the first place. Handy for this sort of thing:

```
my $obj = Some::Module->new;
isa_ok($obj, 'Some::Module');
```

where you'd otherwise have to write

```
my $obj = Some::Module->new;
ok(defined $obj && $obj->isa('Some::Module'));
```

to safeguard against your test script blowing up.

It works on references, too:

```
isa_ok($array_ref, 'ARRAY');
```

The diagnostics of this test normally just refer to 'the object'. If you'd like them to be more specific, you can supply an `$object_name` (for example 'Test customer').

### **pass**

**fail**

```
pass($test_name);
fail($test_name);
```

Sometimes you just want to say that the tests have passed. Usually the case is you've got some complicated condition that is difficult to wedge into an `ok()`. In this case, you can simply use `pass()` (to declare the test ok) or `fail()` (for not ok). They are synonyms for `ok(1)` and `ok(0)`.

Use these very, very, very sparingly.

**Diagnostics**

If you pick the right test function, you'll usually get a good idea of what went wrong when it failed. But sometimes it doesn't work out that way. So here we have ways for you to write your own diagnostic messages which are safer than just `print STDERR`.

**diag**

```
diag(@diagnostic_message);
```

Prints a diagnostic message which is guaranteed not to interfere with test output. Handy for this sort of thing:

```
ok(grep(/foo/, @users), "There's a foo user") or
 diag("Since there's no foo, check that /etc/bar is set up right");
```

which would produce:

```
not ok 42 - There's a foo user
Failed test (foo.t at line 52)
Since there's no foo, check that /etc/bar is set up right.
```

You might remember `ok()` or `diag()` with the mnemonic `open()` or `die()`.

**NOTE** The exact formatting of the diagnostic output is still changing, but it is guaranteed that whatever you throw at it it won't interfere with the test.

**Module tests**

You usually want to test if the module you're testing loads ok, rather than just vomiting if its load fails. For such purposes we have `use_ok` and `require_ok`.

**use\_ok**

```
BEGIN { use_ok($module); }
BEGIN { use_ok($module, @imports); }
```

These simply use the given `$module` and test to make sure the load happened ok. It's recommended that you run `use_ok()` inside a `BEGIN` block so its functions are exported at compile-time and prototypes are properly honored.

If `@imports` are given, they are passed through to the `use`. So this:

```
BEGIN { use_ok('Some::Module', qw(foo bar)) }
```

is like doing this:

```
use Some::Module qw(foo bar);
```

don't try to do this:

```
BEGIN {
 use_ok('Some::Module');
 ...some code that depends on the use...
 ...happening at compile time...
}
```

instead, you want:

```
BEGIN { use_ok('Some::Module') }
BEGIN { ...some code that depends on the use... }
```

### **require\_ok**

```
require_ok($module);
```

Like `use_ok()`, except it requires the `$module`.

### **Conditional tests**

Sometimes running a test under certain conditions will cause the test script to die. A certain function or method isn't implemented (such as `fork()` on MacOS), some resource isn't available (like a net connection) or a module isn't available. In these cases it's necessary to skip tests, or declare that they are supposed to fail but will work in the future (a todo test).

For more details on the mechanics of skip and todo tests see [Test::Harness](#).

The way `Test::More` handles this is with a named block. Basically, a block of tests which can be skipped over or made todo. It's best if I just show you...

### **SKIP: BLOCK**

```
SKIP: {
 skip $why, $how_many if $condition;
 ...normal testing code goes here...
}
```

This declares a block of tests that might be skipped, `$how_many` tests there are, `$why` and under what `$condition` to skip them. An example is the easiest way to illustrate:

```
SKIP: {
 eval { require HTML::Lint };
 skip "HTML::Lint not installed", 2 if $@;
 my $lint = new HTML::Lint;
 ok($lint, "Created object");
 $lint->parse($html);
 is(scalar $lint->errors, 0, "No errors found in HTML");
}
```

If the user does not have `HTML::Lint` installed, the whole block of code *won't be run at all*. `Test::More` will output special ok's which `Test::Harness` interprets as skipped, but passing, tests. It's important that `$how_many` accurately reflects the number of tests in the `SKIP` block so the # of tests run will match up with your plan.

It's perfectly safe to nest `SKIP` blocks. Each `SKIP` block must have the label `SKIP`, or `Test::More` can't work its magic.

You don't skip tests which are failing because there's a bug in your program, or for which you don't yet have code written. For that you use `TODO`. Read on.

### **TODO: BLOCK**

```
TODO: {
 local $TODO = $why if $condition;
 ...normal testing code goes here...
}
```

Declares a block of tests you expect to fail and `$why`. Perhaps it's because you haven't fixed a bug or haven't finished a new feature:



```

TODO: {
 local $TODO = "URI::Geller not finished";

 my $card = "Eight of clubs";
 is(URI::Geller->your_card, $card, 'Is THIS your card?');

 my $spoon;
 URI::Geller->bend_spoon;
 is($spoon, 'bent', "Spoon bending, that's original");
}

```

With a todo block, the tests inside are expected to fail. Test::More will run the tests normally, but print out special flags indicating they are "todo". Test::Harness will interpret failures as being ok. Should anything succeed, it will report it as an unexpected success. You then know the thing you had todo is done and can remove the TODO flag.

The nice part about todo tests, as opposed to simply commenting out a block of tests, is it's like having a programmatic todo list. You know how much work is left to be done, you're aware of what bugs there are, and you'll know immediately when they're fixed.

Once a todo test starts succeeding, simply move it outside the block. When the block is empty, delete it.

### todo\_skip

```

TODO: {
 todo_skip $why, $show_many if $condition;
 ...normal testing code...
}

```

With todo tests, it's best to have the tests actually run. That way you'll know when they start passing. Sometimes this isn't possible. Often a failing test will cause the whole program to die or hang, even inside an eval BLOCK with and using alarm. In these extreme cases you have no choice but to skip over the broken tests entirely.

The syntax and behavior is similar to a SKIP: BLOCK except the tests will be marked as failing but todo. Test::Harness will interpret them as passing.

### When do I use SKIP vs. TODO?

**If it's something the user might not be able to do**, use SKIP. This includes optional modules that aren't installed, running under an OS that doesn't have some feature (like fork() or symlinks), or maybe you need an Internet connection and one isn't available.

**If it's something the programmer hasn't done yet**, use TODO. This is for any code you haven't written yet, or bugs you have yet to fix, but want to put tests in your testing script (always a good idea).

## Comparison functions

Not everything is a simple eq check or regex. There are times you need to see if two arrays are equivalent, for instance. For these instances, Test::More provides a handful of useful functions.

**NOTE** These are NOT well-tested on circular references. Nor am I quite sure what will happen with filehandles.

### is\_deeply

```
is_deeply($this, $that, $test_name);
```

Similar to is(), except that if \$this and \$that are hash or array references, it does a deep comparison walking each data structure to see if they are equivalent. If the two structures are different, it will display the place where they start differing.

Barrie Slaymaker's Test::Differences module provides more in-depth functionality along these lines, and it plays well with Test::More.

**NOTE** Display of scalar refs is not quite 100%

### eq\_array

```
eq_array(\@this, \@that);
```

Checks if two arrays are equivalent. This is a deep check, so multi-level structures are handled correctly.

### eq\_hash

```
eq_hash(\%this, \%that);
```

Determines if the two hashes contain the same keys and values. This is a deep check.

### eq\_set

```
eq_set(\@this, \@that);
```

Similar to eq\_array(), except the order of the elements is **not** important. This is a deep check, but the irrelevancy of order only applies to the top level.

## Extending and Embedding Test::More

Sometimes the Test::More interface isn't quite enough. Fortunately, Test::More is built on top of Test::Builder which provides a single, unified backend for any test library to use. This means two test libraries which both use Test::Builder **can be used together in the same program**.

If you simply want to do a little tweaking of how the tests behave, you can access the underlying Test::Builder object like so:

### builder

```
my $test_builder = Test::More->builder;
```

Returns the Test::Builder object underlying Test::More for you to play with.

## NOTES

Test::More is **explicitly** tested all the way back to perl 5.004.

Test::More is thread-safe for perl 5.8.0 and up.

## BUGS and CAVEATS

Making your own ok()

If you are trying to extend Test::More, don't. Use Test::Builder instead.

The eq\_\* family has some caveats.

Test::Harness upgrades

no\_plan and todo depend on new Test::Harness features and fixes. If you're going to distribute tests that use no\_plan or todo your end-users will have to upgrade Test::Harness to the latest one on CPAN.

If you avoid no\_plan and TODO tests, the stock Test::Harness will work fine.

If you simply depend on Test::More, it's own dependencies will cause a Test::Harness upgrade.

## HISTORY

This is a case of convergent evolution with Joshua Pritikin's Test module. I was largely unaware of its existence when I'd first written my own ok() routines. This module exists because I can't figure out how to easily wedge test names into Test's interface (along with a few other problems).

The goal here is to have a testing utility that's simple to learn, quick to use and difficult to trip yourself up with while still providing more flexibility than the existing Test.pm. As such, the names of the most common routines are kept tiny, special cases and magic side-effects are kept to a minimum. WYSIWYG.

**SEE ALSO**

*Test::Simple* if all this confuses you and you just want to write some tests. You can upgrade to *Test::More* later (it's forward compatible).

*Test::Differences* for more ways to test complex data structures. And it plays well with *Test::More*.

*Test* is the old testing module. Its main benefit is that it has been distributed with Perl since 5.004\_05.

*Test::Harness* for details on how your test results are interpreted by Perl.

*Test::Unit* describes a very featureful unit testing interface.

*Test::Inline* shows the idea of embedded testing.

*SelfTest* is another approach to embedded testing.

**AUTHORS**

Michael G Schwern <schwern@pobox.com> with much inspiration from Joshua Pritikin's *Test* module and lots of help from Barrie Slaymaker, Tony Bowden, chromatic and the perl-qa gang.

**COPYRIGHT**

Copyright 2001 by Michael G Schwern <schwern@pobox.com>.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See <http://www.perl.com/perl/misc/Artistic.html>

**NAME**

Test::Simple – Basic utilities for writing tests.

**SYNOPSIS**

```
use Test::Simple tests => 1;

ok($foo eq $bar, 'foo is bar');
```

**DESCRIPTION**

**\*\* If you are unfamiliar with testing read `Test::Tutorial` first! \*\***

This is an extremely simple, extremely basic module for writing tests suitable for CPAN modules and other pursuits. If you wish to do more complicated testing, use the `Test::More` module (a drop-in replacement for this one).

The basic unit of Perl testing is the `ok`. For each thing you want to test your program will print out an "ok" or "not ok" to indicate pass or fail. You do this with the `ok()` function (see below).

The only other constraint is you must pre-declare how many tests you plan to run. This is in case something goes horribly wrong during the test and your test program aborts, or skips a test or whatever. You do this like so:

```
use Test::Simple tests => 23;
```

You must have a plan.

**ok**

```
ok($foo eq $bar, $name);
ok($foo eq $bar);
```

`ok()` is given an expression (in this case `$foo eq $bar`). If it's true, the test passed. If it's false, it didn't. That's about it.

`ok()` prints out either "ok" or "not ok" along with a test number (it keeps track of that for you).

```
This produces "ok 1 - Hell not yet frozen over" (or not ok)
ok(get_temperature($hell) > 0, 'Hell not yet frozen over');
```

If you provide a `$name`, that will be printed along with the "ok/not ok" to make it easier to find your test when it fails (just search for the name). It also makes it easier for the next guy to understand what your test is for. It's highly recommended you use test names.

All tests are run in scalar context. So this:

```
ok(@stuff, 'I have some stuff');
```

will do what you mean (fail if `stuff` is empty)

`Test::Simple` will start by printing number of tests run in the form "1..M" (so "1..5" means you're going to run 5 tests). This strange format lets `Test::Harness` know how many tests you plan on running in case something goes horribly wrong.

If all your tests passed, `Test::Simple` will exit with zero (which is normal). If anything failed it will exit with how many failed. If you run less (or more) tests than you planned, the missing (or extras) will be considered failures. If no tests were ever run `Test::Simple` will throw a warning and exit with 255. If the test died, even after having successfully completed all its tests, it will still be considered a failure and will exit with 255.

So the exit codes are...

|                  |                                               |
|------------------|-----------------------------------------------|
| 0                | all tests successful                          |
| 255              | test died                                     |
| any other number | how many failed (including missing or extras) |

If you fail more than 254 tests, it will be reported as 254.

This module is by no means trying to be a complete testing system. It's just to get you started. Once you're off the ground its recommended you look at [Test::More](#).

## EXAMPLE

Here's an example of a simple .t file for the fictional Film module.

```
use Test::Simple tests => 5;

use Film; # What you're testing.

my $btaste = Film->new({ Title => 'Bad Taste',
 Director => 'Peter Jackson',
 Rating => 'R',
 NumExplodingSheep => 1
 });

ok(defined($btaste) and ref $btaste eq 'Film', 'new() works');

ok($btaste->Title eq 'Bad Taste', 'Title() get');
ok($btaste->Director eq 'Peter Jackson', 'Director() get');
ok($btaste->Rating eq 'R', 'Rating() get');
ok($btaste->NumExplodingSheep == 1, 'NumExplodingSheep() get');
```

It will produce output like this:

```
1..5
ok 1 - new() works
ok 2 - Title() get
ok 3 - Director() get
not ok 4 - Rating() get
Failed test (t/film.t at line 14)
ok 5 - NumExplodingSheep() get
Looks like you failed 1 tests of 5
```

Indicating the `Film::Rating()` method is broken.

## CAVEATS

`Test::Simple` will only report a maximum of 254 failures in its exit code. If this is a problem, you probably have a huge test script. Split it into multiple files. (Otherwise blame the Unix folks for using an unsigned short integer as the exit status).

Because VMS's exit codes are much, much different than the rest of the universe, and perl does horrible mangling to them that gets in my way, it works like this on VMS.

|   |             |                      |
|---|-------------|----------------------|
| 0 | SS\$_NORMAL | all tests successful |
| 4 | SS\$_ABORT  | something went wrong |

Unfortunately, I can't differentiate any further.

## NOTES

`Test::Simple` is **explicitly** tested all the way back to perl 5.004.

`Test::Simple` is thread-safe in perl 5.8.0 and up.

## HISTORY

This module was conceived while talking with Tony Bowden in his kitchen one night about the problems I was having writing some really complicated feature into the new Testing module. He observed that the main problem is not dealing with these edge cases but that people hate to write tests **at all**. What was needed was a dead simple module that took all the hard work out of testing and was really, really easy to learn. Paul Johnson simultaneously had this idea (unfortunately, he wasn't in Tony's kitchen). This is it.

**SEE ALSO***Test::More*

More testing functions! Once you outgrow `Test::Simple`, look at `Test::More`. `Test::Simple` is 100% forward compatible with `Test::More` (i.e. you can just use `Test::More` instead of `Test::Simple` in your programs and things will still work).

*Test*

The original Perl testing module.

*Test::Unit*

Elaborate unit testing.

*Test::Inline*, *SelfTest*

Embed tests in your code!

*Test::Harness*

Interprets the output of your test program.

**AUTHORS**

Idea by Tony Bowden and Paul Johnson, code by Michael G Schwern <schwern@pobox.com>, wardrobe by Calvin Klein.

**COPYRIGHT**

Copyright 2001 by Michael G Schwern <schwern@pobox.com>.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See <http://www.perl.com/perl/misc/Artistic.html>

**NAME**

Test::Tutorial – A tutorial about writing really basic tests

**DESCRIPTION**

*AHHHHHHH!!!! NOT TESTING! Anything but testing! Beat me, whip me, send me to Detroit, but don't make me write tests!*

*\*sob\**

*Besides, I don't know how to write the damned things.*

Is this you? Is writing tests right up there with writing documentation and having your fingernails pulled out? Did you open up a test and read

```
We start with some black magic
and decide that's quite enough for you?
```

It's ok. That's all gone now. We've done all the black magic for you. And here are the tricks...

**Nuts and bolts of testing.**

Here's the most basic test program.

```
#!/usr/bin/perl -w
print "1..1\n";
print 1 + 1 == 2 ? "ok 1\n" : "not ok 1\n";
```

since 1 + 1 is 2, it prints:

```
1..1
ok 1
```

What this says is: 1..1 "I'm going to run one test." [1] ok 1 "The first test passed". And that's about all magic there is to testing. Your basic unit of testing is the *ok*. For each thing you test, an *ok* is printed. Simple. **Test::Harness** interprets your test results to determine if you succeeded or failed (more on that later).

Writing all these print statements rapidly gets tedious. Fortunately, there's **Test::Simple**. It has one function, `ok()`.

```
#!/usr/bin/perl -w
use Test::Simple tests => 1;
ok(1 + 1 == 2);
```

and that does the same thing as the code above. `ok()` is the backbone of Perl testing, and we'll be using it instead of roll-your-own from here on. If `ok()` gets a true value, the test passes. False, it fails.

```
#!/usr/bin/perl -w
use Test::Simple tests => 2;
ok(1 + 1 == 2);
ok(2 + 2 == 5);
```

from that comes

```
1..2
ok 1
not ok 2
Failed test (test.pl at line 5)
Looks like you failed 1 tests of 2.
```

1..2 "I'm going to run two tests." This number is used to ensure your test program ran all the way through and didn't die or skip some tests. `ok 1 "The first test passed."` `not ok 2 "The second test failed"`. `Test::Simple` helpfully prints out some extra commentary about your tests.

It's not scary. Come, hold my hand. We're going to give an example of testing a module. For our example, we'll be testing a date library, **Date::ICal**. It's on CPAN, so download a copy and follow along. [2]

### Where to start?

This is the hardest part of testing, where do you start? People often get overwhelmed at the apparent enormity of the task of testing a whole module. Best place to start is at the beginning. `Date::ICal` is an object-oriented module, and that means you start by making an object. So we test `new()`.

```
#!/usr/bin/perl -w

use Test::Simple tests => 2;

use Date::ICal;

my $ical = Date::ICal->new; # create an object
ok(defined $ical); # check that we got something
ok($ical->isa('Date::ICal')); # and it's the right class
```

run that and you should get:

```
1..2
ok 1
ok 2
```

congratulations, you've written your first useful test.

### Names

That output isn't terribly descriptive, is it? When you have two tests you can figure out which one is #2, but what if you have 102?

Each test can be given a little descriptive name as the second argument to `ok()`.

```
use Test::Simple tests => 2;

ok(defined $ical, 'new() returned something');
ok($ical->isa('Date::ICal'), " and it's the right class");
```

So now you'd see...

```
1..2
ok 1 - new() returned something
ok 2 - and it's the right class
```

### Test the manual

Simplest way to build up a decent testing suite is to just test what the manual says it does. [3] Let's pull something out of the [Date::ICal/SYNOPSIS](#) and test that all its bits work.

```
#!/usr/bin/perl -w

use Test::Simple tests => 8;

use Date::ICal;

$ical = Date::ICal->new(year => 1964, month => 10, day => 16,
 hour => 16, min => 12, sec => 47,
 tz => '0530');

ok(defined $ical, 'new() returned something');
ok($ical->isa('Date::ICal'), " and it's the right class");
ok($ical->sec == 47, ' sec()');
```



```

ok($ical->min == 12, ' min()');
ok($ical->hour == 16, ' hour()');
ok($ical->day == 17, ' day()');
ok($ical->month == 10, ' month()');
ok($ical->year == 1964, ' year()');

```

run that and you get:

```

1..8
ok 1 - new() returned something
ok 2 - and it's the right class
ok 3 - sec()
ok 4 - min()
ok 5 - hour()
not ok 6 - day()
Failed test (- at line 16)
ok 7 - month()
ok 8 - year()
Looks like you failed 1 tests of 8.

```

Whoops, a failure! [4] `Test::Simple` helpfully lets us know on what line the failure occurred, but not much else. We were supposed to get 17, but we didn't. What did we get?? Dunno. We'll have to re-run the test in the debugger or throw in some print statements to find out.

Instead, we'll switch from **Test::Simple** to **Test::More**. **Test::More** does everything **Test::Simple** does, and more! In fact, `Test::More` does things *exactly* the way `Test::Simple` does. You can literally swap `Test::Simple` out and put `Test::More` in its place. That's just what we're going to do.

`Test::More` does more than `Test::Simple`. The most important difference at this point is it provides more informative ways to say "ok". Although you can write almost any test with a generic `ok()`, it can't tell you what went wrong. Instead, we'll use the `is()` function, which lets us declare that something is supposed to be the same as something else:

```

#!/usr/bin/perl -w

use Test::More tests => 8;

use Date::ICal;

$ical = Date::ICal->new(year => 1964, month => 10, day => 16,
 hour => 16, min => 12, sec => 47,
 tz => '0530');

ok(defined $ical, 'new() returned something');
ok($ical->isa('Date::ICal'), " and it's the right class");
is($ical->sec, 47, ' sec()');
is($ical->min, 12, ' min()');
is($ical->hour, 16, ' hour()');
is($ical->day, 17, ' day()');
is($ical->month, 10, ' month()');
is($ical->year, 1964, ' year()');

```

"Is `$ical->sec 47`?" "Is `$ical->min 12`?" With `is()` in place, you get some more information

```

1..8
ok 1 - new() returned something
ok 2 - and it's the right class
ok 3 - sec()
ok 4 - min()
ok 5 - hour()

```

```

not ok 6 - day()
Failed test (- at line 16)
got: '16'
expected: '17'
ok 7 - month()
ok 8 - year()
Looks like you failed 1 tests of 8.

```

letting us know that `$ical->day` returned 16, but we expected 17. A quick check shows that the code is working fine, we made a mistake when writing up the tests. Just change it to:

```
is($ical->day, 16, ' day()');
```

and everything works.

So any time you're doing a "this equals that" sort of test, use `is()`. It even works on arrays. The test is always in scalar context, so you can test how many elements are in a list this way. [5]

```
is(@foo, 5, 'foo has 5 elements');
```

### Sometimes the tests are wrong

Which brings us to a very important lesson. Code has bugs. Tests are code. Ergo, tests have bugs. A failing test could mean a bug in the code, but don't discount the possibility that the test is wrong.

On the flip side, don't be tempted to prematurely declare a test incorrect just because you're having trouble finding the bug. Invalidating a test isn't something to be taken lightly, and don't use it as a cop out to avoid work.

### Testing lots of values

We're going to be wanting to test a lot of dates here, trying to trick the code with lots of different edge cases. Does it work before 1970? After 2038? Before 1904? Do years after 10,000 give it trouble? Does it get leap years right? We could keep repeating the code above, or we could set up a little try/expect loop.

```

use Test::More tests => 32;
use Date::ICal;

my %ICal_Dates = (
 # An ICal string And the year, month, date
 # hour, minute and second we expect.
 '19971024T120000' => # from the docs.
 [1997, 10, 24, 12, 0, 0],
 '20390123T232832' => # after the Unix epoch
 [2039, 1, 23, 23, 28, 32],
 '19671225T000000' => # before the Unix epoch
 [1967, 12, 25, 0, 0, 0],
 '18990505T232323' => # before the MacOS epoch
 [1899, 5, 5, 23, 23, 23],
);

while(my($ical_str, $expect) = each %ICal_Dates) {
 my $ical = Date::ICal->new(ical => $ical_str);

 ok(defined $ical, "new(ical => '$ical_str')");
 ok($ical->isa('Date::ICal'), " and it's the right class");

 is($ical->year, $expect->[0], ' year()');
 is($ical->month, $expect->[1], ' month()');
 is($ical->day, $expect->[2], ' day()');
 is($ical->hour, $expect->[3], ' hour()');
 is($ical->min, $expect->[4], ' min()');
 is($ical->sec, $expect->[5], ' sec()');
}

```

```
}
```

So now we can test bunches of dates by just adding them to %ICal\_Dates. Now that it's less work to test with more dates, you'll be inclined to just throw more in as you think of them. Only problem is, every time we add to that we have to keep adjusting the use `Test::More tests => ##` line. That can rapidly get annoying. Instead we use `no_plan`. This means we're just running some tests, don't know how many. [6]

```
use Test::More 'no_plan'; # instead of tests => 32
```

now we can just add tests and not have to do all sorts of math to figure out how many we're running.

### Informative names

Take a look at this line here

```
ok(defined $ical, "new(ical => '$ical_str')");
```

we've added more detail about what we're testing and the ICal string itself we're trying out to the name. So you get results like:

```
ok 25 - new(ical => '19971024T120000')
ok 26 - and it's the right class
ok 27 - year()
ok 28 - month()
ok 29 - day()
ok 30 - hour()
ok 31 - min()
ok 32 - sec()
```

if something in there fails, you'll know which one it was and that will make tracking down the problem easier. So try to put a bit of debugging information into the test names.

Describe what the tests test, to make debugging a failed test easier for you or for the next person who runs your test.

### Skipping tests

Poking around in the existing Date::ICal tests, I found this in *t/01sanity.t* [7]

```
#!/usr/bin/perl -w

use Test::More tests => 7;
use Date::ICal;

Make sure epoch time is being handled sanely.
my $t1 = Date::ICal->new(epoch => 0);
is($t1->epoch, 0, "Epoch time of 0");

XXX This will only work on unix systems.
is($t1->ical, '19700101Z', "epoch to ical");

is($t1->year, 1970, "year()");
is($t1->month, 1, "month()");
is($t1->day, 1, "day()");

like the tests above, but starting with ical instead of epoch
my $t2 = Date::ICal->new(ical => '19700101Z');
is($t2->ical, '19700101Z', "Start of epoch in ICal notation");

is($t2->epoch, 0, "and back to ICal");
```

The beginning of the epoch is different on most non-Unix operating systems [8]. Even though Perl smooths out the differences for the most part, certain ports do it differently. MacPerl is one off the top of my head. [9] We *know* this will never work on MacOS. So rather than just putting a comment in the test, we can

explicitly say it's never going to work and skip the test.

```
use Test::More tests => 7;
use Date::ICal;

Make sure epoch time is being handled sanely.
my $t1 = Date::ICal->new(epoch => 0);
is($t1->epoch, 0, "Epoch time of 0");

SKIP: {
 skip('epoch to ICal not working on MacOS', 6)
 if $^O eq 'MacOS';

 is($t1->ical, '19700101Z', " epoch to ical");

 is($t1->year, 1970, " year()");
 is($t1->month, 1, " month()");
 is($t1->day, 1, " day()");

 # like the tests above, but starting with ical instead of epoch
 my $t2 = Date::ICal->new(ical => '19700101Z');
 is($t2->ical, '19700101Z', "Start of epoch in ICal notation");
 is($t2->epoch, 0, " and back to ICal");
}
```

A little bit of magic happens here. When running on anything but MacOS, all the tests run normally. But when on MacOS, `skip()` causes the entire contents of the SKIP block to be jumped over. It's never run. Instead, it prints special output that tells Test::Harness that the tests have been skipped.

```
1..7
ok 1 - Epoch time of 0
ok 2 # skip epoch to ICal not working on MacOS
ok 3 # skip epoch to ICal not working on MacOS
ok 4 # skip epoch to ICal not working on MacOS
ok 5 # skip epoch to ICal not working on MacOS
ok 6 # skip epoch to ICal not working on MacOS
ok 7 # skip epoch to ICal not working on MacOS
```

This means your tests won't fail on MacOS. This means less emails from MacPerl users telling you about failing tests that you know will never work. You've got to be careful with skip tests. These are for tests which don't work and *never will*. It is not for skipping genuine bugs (we'll get to that in a moment).

The tests are wholly and completely skipped. [10] This will work.

```
SKIP: {
 skip("I don't wanna die!");
 die, die, die, die, die;
}
```

## Todo tests

Thumbing through the Date::ICal man page, I came across this:

```
ical

 $ical_string = $ical->ical;

Retrieves, or sets, the date on the object, using any
valid ICal date/time string.
```

"Retrieves or sets". Hmmm, didn't see a test for using `ical()` to set the date in the Date::ICal test suite. So I'll write one.

```

use Test::More tests => 1;

my $ical = Date::ICal->new;
$ical->ical('20201231Z');
is($ical->ical, '20201231Z', 'Setting via ical()');

```

run that and I get

```

1..1
not ok 1 - Setting via ical()
Failed test (- at line 6)
got: '20010814T233649Z'
expected: '20201231Z'
Looks like you failed 1 tests of 1.

```

Whoops! Looks like it's unimplemented. Let's assume we don't have the time to fix this. [11] Normally, you'd just comment out the test and put a note in a todo list somewhere. Instead, we're going to explicitly state "this test will fail" by wrapping it in a TODO block.

```

use Test::More tests => 1;

TODO: {
 local $TODO = 'ical($ical) not yet implemented';

 my $ical = Date::ICal->new;
 $ical->ical('20201231Z');

 is($ical->ical, '20201231Z', 'Setting via ical()');
}

```

Now when you run, it's a little different:

```

1..1
not ok 1 - Setting via ical() # TODO ical($ical) not yet implemented
got: '20010822T201551Z'
expected: '20201231Z'

```

Test::More doesn't say "Looks like you failed 1 tests of 1". That '# TODO' tells Test::Harness "this is supposed to fail" and it treats a failure as a successful test. So you can write tests even before you've fixed the underlying code.

If a TODO test passes, Test::Harness will report it "UNEXPECTEDLY SUCCEEDED". When that happens, you simply remove the TODO block with `local $TODO` and turn it into a real test.

### Testing with taint mode.

Taint mode is a funny thing. It's the globeleest of all global features. Once you turn it on it effects *all* code in your program and *all* modules used (and all the modules they use). If a single piece of code isn't taint clean, the whole thing explodes. With that in mind, it's very important to ensure your module works under taint mode.

It's very simple to have your tests run under taint mode. Just throw a `-T` into the `#!` line. Test::Harness will read the switches in `#!` and use them to run your tests.

```

#!/usr/bin/perl -Tw

use Test::More 'no_plan';

...test normally here...

```

So when you say `make test` it will be run with taint mode and warnings on.

## FOOTNOTES

- 1 The first number doesn't really mean anything, but it has to be 1. It's the second number that's important.
- 2 For those following along at home, I'm using version 1.31. It has some bugs, which is good — we'll uncover them with our tests.
- 3 You can actually take this one step further and test the manual itself. Have a look at **Test::Inline** (formerly **Pod::Tests**).
- 4 Yes, there's a mistake in the test suite. What! Me, contrived?
- 5 We'll get to testing the contents of lists later.
- 6 But what happens if your test program dies halfway through?! Since we didn't say how many tests we're going to run, how can we know it failed? No problem, **Test::More** employs some magic to catch that death and turn the test into a failure, even if every test passed up to that point.
- 7 I cleaned it up a little.
- 8 Most Operating Systems record time as the number of seconds since a certain date. This date is the beginning of the epoch. Unix's starts at midnight January 1st, 1970 GMT.
- 9 MacOS's epoch is midnight January 1st, 1904. VMS's is midnight, November 17th, 1858, but **vmperl** emulates the Unix epoch so it's not a problem.
- 10 As long as the code inside the **SKIP** block at least compiles. Please don't ask how. No, it's not a filter.
- 11 Do NOT be tempted to use **TODO** tests as a way to avoid fixing simple bugs!

## AUTHORS

Michael G Schwern <schwern@pobox.com> and the perl-qa dancers!

## COPYRIGHT

Copyright 2001 by Michael G Schwern <schwern@pobox.com>.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in these files are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

**NAME**

abbrev – create an abbreviation table from a list

**SYNOPSIS**

```
use Text::Abbrev;
abbrev $hashref, LIST
```

**DESCRIPTION**

Stores all unambiguous truncations of each element of `LIST` as keys in the associative array referenced by `$hashref`. The values are the original list elements.

**EXAMPLE**

```
$hashref = abbrev qw(list edit send abort gripe);
%hash = abbrev qw(list edit send abort gripe);
abbrev $hashref, qw(list edit send abort gripe);
abbrev(*hash, qw(list edit send abort gripe));
```

**NAME**

Text::Balanced – Extract delimited text sequences from strings.

**SYNOPSIS**

```
use Text::Balanced qw (
 extract_delimited
 extract_bracketed
 extract_quotelike
 extract_codeblock
 extract_variable
 extract_tagged
 extract_multiple

 gen_delimited_pat
 gen_extract_tagged
);

Extract the initial substring of $text that is delimited by
two (unescaped) instances of the first character in $delim.

($extracted, $remainder) = extract_delimited($text,$delim);

Extract the initial substring of $text that is bracketed
with a delimiter(s) specified by $delim (where the string
in $delim contains one or more of '(){}[]<>').

($extracted, $remainder) = extract_bracketed($text,$delim);

Extract the initial substring of $text that is bounded by
an HTML/XML tag.

($extracted, $remainder) = extract_tagged($text);

Extract the initial substring of $text that is bounded by
a C<BEGIN>...C<END> pair. Don't allow nested C<BEGIN> tags

($extracted, $remainder) =
 extract_tagged($text,"BEGIN","END",undef,{bad=>["BEGIN"]});

Extract the initial substring of $text that represents a
Perl "quote or quote-like operation"

($extracted, $remainder) = extract_quotelike($text);

Extract the initial substring of $text that represents a block
of Perl code, bracketed by any of character(s) specified by $delim
(where the string $delim contains one or more of '(){}[]<>').

($extracted, $remainder) = extract_codeblock($text,$delim);

Extract the initial substrings of $text that would be extracted by
one or more sequential applications of the specified functions
or regular expressions

@extracted = extract_multiple($text,
 [\&extract_bracketed,
 \&extract_quotelike,
 \&some_other_extractor_sub,
 qr/[xyz]*/,
 'literal',
]);
```



# Create a string representing an optimized pattern (a la Friedl) # that matches a substring delimited by any of the specified characters # (in this case: any type of quote or a slash)

```
$patstring = gen_delimited_pat(q{'"\'/});
```

# Generate a reference to an anonymous sub that is just like `extract_tagged` # but pre-compiled and optimized for a specific pair of tags, and consequently # much faster (i.e. 3 times faster). It uses `qr//` for better performance on # repeated calls, so it only works under Perl 5.005 or later.

```
$extract_head = gen_extract_tagged('<HEAD>', '</HEAD>');
```

```
($extracted, $remainder) = $extract_head->($text);
```

## DESCRIPTION

The various `extract_...` subroutines may be used to extract a delimited string (possibly after skipping a specified prefix string). The search for the string always begins at the current `pos` location of the string's variable (or at index zero, if no `pos` position is defined).

### General behaviour in list contexts

In a list context, all the subroutines return a list, the first three elements of which are always:

- [0] The extracted string, including the specified delimiters. If the extraction fails an empty string is returned.
- [1] The remainder of the input string (i.e. the characters after the extracted string). On failure, the entire string is returned.
- [2] The skipped prefix (i.e. the characters before the extracted string). On failure, the empty string is returned.

Note that in a list context, the contents of the original input text (the first argument) are not modified in any way.

However, if the input text was passed in a variable, that variable's `pos` value is updated to point at the first character after the extracted text. That means that in a list context the various subroutines can be used much like regular expressions. For example:

```
while ($next = (extract_quotelike($text))[0])
{
 # process next quote-like (in $next)
}
```

### General behaviour in scalar and void contexts

In a scalar context, the extracted string is returned, having first been removed from the input text. Thus, the following code also processes each quote-like operation, but actually removes them from `$text`:

```
while ($next = extract_quotelike($text))
{
 # process next quote-like (in $next)
}
```

Note that if the input text is a read-only string (i.e. a literal), no attempt is made to remove the extracted text.

In a void context the behaviour of the extraction subroutines is exactly the same as in a scalar context, except (of course) that the extracted substring is not returned.

### A note about prefixes

Prefix patterns are matched without any trailing modifiers (`/gimsx` etc.) This can bite you if you're expecting a prefix specification like `'.*(?=<H1)'` to skip everything up to the first `<H1` tag. Such a prefix pattern will only succeed if the `<H1` tag is on the current line, since `.` normally doesn't match newlines.

To overcome this limitation, you need to turn on /s matching within the prefix pattern, using the (?s) directive: '(?s).\*(?=<H1)'

```
extract delimited
```

The `extract_delimited` function formalizes the common idiom of extracting a single-character-delimited substring from the start of a string. For example, to extract a single-quote delimited string, the following code is typically used:

```
($remainder = $text) =~ s/\A('(\.|\[^\])*')//s;
$extracted = $1;
```

but with `extract delimited` it can be simplified to:

```
($extracted,$remainder) = extract delimited($text, "'");
```

`extract_delimited` takes up to four scalars (the input text, the delimiters, a prefix pattern to be skipped, and any escape characters) and extracts the initial substring of the text that is appropriately delimited. If the delimiter string has multiple characters, the first one encountered in the text is taken to delimit the substring. The third argument specifies a prefix pattern that is to be skipped (but must be present!) before the substring is extracted. The final argument specifies the escape character to be used for each delimiter.

All arguments are optional. If the escape characters are not specified, every delimiter is escaped with a backslash (\). If the prefix is not specified, the pattern `\s*` – optional whitespace – is used. If the delimiter set is also not specified, the set `/[\"'`]/` is used. If the text to be processed is not specified either, `$` is used.

In list context, `extract_delimited` returns an array of three elements, the extracted substring (*including the surrounding delimiters*), the remainder of the text, and the skipped prefix (if any). If a suitable delimited substring is not found, the first element of the array is the empty string, the second is the complete original text, and the prefix returned in the third element is an empty string.

In a scalar context, just the extracted substring is returned. In a void context, the extracted substring (and any prefix) are simply removed from the beginning of the first argument.

Examples:

```
Remove a single-quoted substring from the very beginning of $text:
 $substring = extract_delimited($text, "'", '');

Remove a single-quoted Pascalish substring (i.e. one in which
doubling the quote character escapes it) from the very
beginning of $text:
 $substring = extract_delimited($text, "'", '"');

Extract a single- or double- quoted substring from the
beginning of $text, optionally after some whitespace
(note the list context to protect $text from modification):
 ($substring) = extract_delimited $text, q{"'"};

Delete the substring delimited by the first '/' in $text:
 $text = join '', (extract_delimited($text, '/', '['^/]*')[2,1]);
```

Note that this last example is *not* the same as deleting the first quote-like pattern. For instance, if `$text` contained the string:

```
"if ('./cmd' =~ m/$UNIXCMD/s) { $cmd = $1; }"
```

then after the deletion it would contain:

```
"if ('. $UNIXCMD/s) { $cmd = $1; }"
```

not:

```
"if ('./cmd' =~ ms) { $cmd = $1; }"
```

See ["extract\\_quotelike"](#) for a (partial) solution to this problem.

### extract\_bracketed

Like `extract_delimited`, the `extract_bracketed` function takes up to three optional scalar arguments: a string to extract from, a delimiter specifier, and a prefix pattern. As before, a missing prefix defaults to optional whitespace and a missing text defaults to `$_`. However, a missing delimiter specifier defaults to `'{ } ( ) [ ] < >'` (see below).

`extract_bracketed` extracts a balanced-bracket-delimited substring (using any one (or more) of the user-specified delimiter brackets: `'(..)', '{..}', '[..]',` or `'<..'`). Optionally it will also respect quoted unbalanced brackets (see below).

A "delimiter bracket" is a bracket in list of delimiters passed as `extract_bracketed`'s second argument. Delimiter brackets are specified by giving either the left or right (or both!) versions of the required bracket(s). Note that the order in which two or more delimiter brackets are specified is not significant.

A "balanced-bracket-delimited substring" is a substring bounded by matched brackets, such that any other (left or right) delimiter bracket *within* the substring is also matched by an opposite (right or left) delimiter bracket *at the same level of nesting*. Any type of bracket not in the delimiter list is treated as an ordinary character.

In other words, each type of bracket specified as a delimiter must be balanced and correctly nested within the substring, and any other kind of ("non-delimiter") bracket in the substring is ignored.

For example, given the string:

```
$text = "{ an '[irregularly :-() {} parenthesized >:-)' string }";
```

then a call to `extract_bracketed` in a list context:

```
@result = extract_bracketed($text, '{ }');
```

would return:

```
("{ an '[irregularly :-() {} parenthesized >:-)' string }" , "" , "")
```

since both sets of `'{ .. }'` brackets are properly nested and evenly balanced. (In a scalar context just the first element of the array would be returned. In a void context, `$text` would be replaced by an empty string.)

Likewise the call in:

```
@result = extract_bracketed($text, '[]');
```

would return the same result, since all sets of both types of specified delimiter brackets are correctly nested and balanced.

However, the call in:

```
@result = extract_bracketed($text, '{ (<');
```

would fail, returning:

```
(undef , "{ an '[irregularly :-() {} parenthesized >:-)' string }");
```

because the embedded pairs of `'(..)'`s and `'[..]'`s are "cross-nested" and the embedded `'>'` is unbalanced. (In a scalar context, this call would return an empty string. In a void context, `$text` would be unchanged.)

Note that the embedded single-quotes in the string don't help in this case, since they have not been specified

as acceptable delimiters and are therefore treated as non-delimiter characters (and ignored).

However, if a particular species of quote character is included in the delimiter specification, then that type of quote will be correctly handled. for example, if `$text` is:

```
$text = '>>>">link';
```

then

```
@result = extract_bracketed($text, '<">');
```

returns:

```
('>>>">', 'link', "")
```

as expected. Without the specification of `"` as an embedded quoter:

```
@result = extract_bracketed($text, '<>');
```

the result would be:

```
('', '>>>>">link', "")
```

In addition to the quote delimiters `'`, `"`, and ```, full Perl quote-like quoting (i.e. `q{string}`, `qq{string}`, etc) can be specified by including the letter `'q'` as a delimiter. Hence:

```
@result = extract_bracketed($text, '<q>');
```

would correctly match something like this:

```
$text = '<leftop: conj /and/ conj>';
```

See also: `"extract_quotelike"` and `"extract_codeblock"`.

### **extract\_tagged**

`extract_tagged` extracts and segments text between (balanced) specified tags.

The subroutine takes up to five optional arguments:

1. A string to be processed (`$_` if the string is omitted or `undef`)
2. A string specifying a pattern to be matched as the opening tag. If the pattern string is omitted (or `undef`) then a pattern that matches any standard HTML/XML tag is used.
3. A string specifying a pattern to be matched at the closing tag. If the pattern string is omitted (or `undef`) then the closing tag is constructed by inserting a `/` after any leading bracket characters in the actual opening tag that was matched (*not* the pattern that matched the tag). For example, if the opening tag pattern is specified as `'{\w+}'` and actually matched the opening tag `"{DATA}"`, then the constructed closing tag would be `"{/DATA}"`.
4. A string specifying a pattern to be matched as a prefix (which is to be skipped). If omitted, optional whitespace is skipped.
5. A hash reference containing various parsing options (see below)

The various options that can be specified are:

```
reject => $listref
```

The list reference contains one or more strings specifying patterns that must *not* appear within the tagged text.

For example, to extract an HTML link (which should not contain nested links) use:

```
extract_tagged($text, '<A>', '', undef, {reject => ['<A>']});
```

**ignore => \$listref**

The list reference contains one or more strings specifying patterns that are *not* to be treated as nested tags within the tagged text (even if they would match the start tag pattern).

For example, to extract an arbitrary XML tag, but ignore "empty" elements:

```
extract_tagged($text, undef, undef, undef, {ignore => ['<[^>]*>']});
```

(also see *["gen\\_delimited\\_pat"](#)* below).

**fail => \$str**

The `fail` option indicates the action to be taken if a matching end tag is not encountered (i.e. before the end of the string or some `reject` pattern matches). By default, a failure to match a closing tag causes `extract_tagged` to immediately fail.

However, if the string value associated with `<reject` is "MAX", then `extract_tagged` returns the complete text up to the point of failure. If the string is "PARA", `extract_tagged` returns only the first paragraph after the tag (up to the first line that is either empty or contains only whitespace characters). If the string is "", the default behaviour (i.e. failure) is reinstated.

For example, suppose the start tag `/para` introduces a paragraph, which then continues until the next `/endpara` tag or until another `/para` tag is encountered:

```
$text = "/para line 1\n\nline 3\n/para line 4";
extract_tagged($text, '/para', '/endpara', undef,
 {reject => '/para', fail => MAX });

EXTRACTED: "/para line 1\n\nline 3\n"
```

Suppose instead, that if no matching `/endpara` tag is found, the `/para` tag refers only to the immediately following paragraph:

```
$text = "/para line 1\n\nline 3\n/para line 4";
extract_tagged($text, '/para', '/endpara', undef,
 {reject => '/para', fail => MAX });

EXTRACTED: "/para line 1\n"
```

Note that the specified `fail` behaviour applies to nested tags as well.

On success in a list context, an array of 6 elements is returned. The elements are:

- [0] the extracted tagged substring (including the outermost tags),
- [1] the remainder of the input text,
- [2] the prefix substring (if any),
- [3] the opening tag
- [4] the text between the opening and closing tags
- [5] the closing tag (or "" if no closing tag was found)

On failure, all of these values (except the remaining text) are `undef`.

In a scalar context, `extract_tagged` returns just the complete substring that matched a tagged text (including the start and end tags). `undef` is returned on failure. In addition, the original input text has the returned substring (and any prefix) removed from it.

In a void context, the input text just has the matched substring (and any specified prefix) removed.

**gen\_extract\_tagged**

(Note: This subroutine is only available under Perl5.005)

`gen_extract_tagged` generates a new anonymous subroutine which extracts text between (balanced) specified tags. In other words, it generates a function identical in function to `extract_tagged`.

The difference between `extract_tagged` and the anonymous subroutines generated by `gen_extract_tagged`, is that those generated subroutines:

- do not have to reparse tag specification or parsing options every time they are called (whereas `extract_tagged` has to effectively rebuild its tag parser on every call);
- make use of the new `qr//` construct to pre-compile the regexes they use (whereas `extract_tagged` uses standard string variable interpolation to create tag-matching patterns).

The subroutine takes up to four optional arguments (the same set as `extract_tagged` except for the string to be processed). It returns a reference to a subroutine which in turn takes a single argument (the text to be extracted from).

In other words, the implementation of `extract_tagged` is exactly equivalent to:

```
sub extract_tagged
{
 my $text = shift;
 $extractor = gen_extract_tagged(@_);
 return $extractor->($text);
}
```

(although `extract_tagged` is not currently implemented that way, in order to preserve pre-5.005 compatibility).

Using `gen_extract_tagged` to create extraction functions for specific tags is a good idea if those functions are going to be called more than once, since their performance is typically twice as good as the more general-purpose `extract_tagged`.

**extract\_quotelike**

`extract_quotelike` attempts to recognize, extract, and segment any one of the various Perl quotes and quotelike operators (see [perlop\(3\)](#)) Nested backslashed delimiters, embedded balanced bracket delimiters (for the quotelike operators), and trailing modifiers are all caught. For example, in:

```
extract_quotelike 'q # an octothorpe: \# (not the end of the q!) #'
extract_quotelike ' "You said, \"Use sed\".\" '
extract_quotelike ' s{ ([A-Z]{1,8}\\. [A-Z]{3}) } /\L$1\E/; '
extract_quotelike ' tr/\\\\/\\\\\\\\\\\\/ds; '
```

the full Perl quotelike operations are all extracted correctly.

Note too that, when using the `/x` modifier on a regex, any comment containing the current pattern delimiter will cause the regex to be immediately terminated. In other words:

```
'm /
 (?i) # CASE INSENSITIVE
 [a-z_] # LEADING ALPHABETIC/UNDERSCORE
 [a-z0-9]* # FOLLOWED BY ANY NUMBER OF ALPHANUMERICS
/x'
```

will be extracted as if it were:

```
'm /
 (?i) # CASE INSENSITIVE
```

```
[a-z_] # LEADING ALPHABETIC/'
```

This behaviour is identical to that of the actual compiler.

`extract_quotelike` takes two arguments: the text to be processed and a prefix to be matched at the very beginning of the text. If no prefix is specified, optional whitespace is the default. If no text is given, `$_` is used.

In a list context, an array of 11 elements is returned. The elements are:

- [0] the extracted quotelike substring (including trailing modifiers),
- [1] the remainder of the input text,
- [2] the prefix substring (if any),
- [3] the name of the quotelike operator (if any),
- [4] the left delimiter of the first block of the operation,
- [5] the text of the first block of the operation (that is, the contents of a quote, the regex of a match or substitution or the target list of a translation),
- [6] the right delimiter of the first block of the operation,
- [7] the left delimiter of the second block of the operation (that is, if it is an `s`, `tr`, or `y`),
- [8] the text of the second block of the operation (that is, the replacement of a substitution or the translation list of a translation),
- [9] the right delimiter of the second block of the operation (if any),
- [10] the trailing modifiers on the operation (if any).

For each of the fields marked "(if any)" the default value on success is an empty string. On failure, all of these values (except the remaining text) are `undef`.

In a scalar context, `extract_quotelike` returns just the complete substring that matched a quotelike operation (or `undef` on failure). In a scalar or void context, the input text has the same substring (and any specified prefix) removed.

Examples:

```
Remove the first quotelike literal that appears in text
 $quotelike = extract_quotelike($text, '.*?');

Replace one or more leading whitespace-separated quotelike
literals in $_ with "<QLL>"
 do { $_ = join '<QLL>', (extract_quotelike)[2,1] } until $@;

Isolate the search pattern in a quotelike operation from $text
 ($op,$pat) = (extract_quotelike $text)[3,5];
 if ($op =~ /[ms]/)
 {
 print "search pattern: $pat\n";
 }
 else
 {
 print "$op is not a pattern matching operation\n";
 }
}
```

**extract\_quotelike and "here documents"**

`extract_quotelike` can successfully extract "here documents" from an input string, but with an important caveat in list contexts.

Unlike other types of quote-like literals, a here document is rarely a contiguous substring. For example, a typical piece of code using here document might look like this:

```
<<'EOMSG' || die;
This is the message.
EOMSG
exit;
```

Given this as an input string in a scalar context, `extract_quotelike` would correctly return the string "<<'EOMSG'\nThis is the message.\nEOMSG", leaving the string " || die;\nexit;" in the original variable. In other words, the two separate pieces of the here document are successfully extracted and concatenated.

In a list context, `extract_quotelike` would return the list

- [0] "<<'EOMSG'\nThis is the message.\nEOMSG\n" (i.e. the full extracted here document, including fore and aft delimiters),
- [1] " || die;\nexit;" (i.e. the remainder of the input text, concatenated),
- [2] "" (i.e. the prefix substring — trivial in this case),
- [3] "<<" (i.e. the "name" of the quotelike operator)
- [4] "'EOMSG'" (i.e. the left delimiter of the here document, including any quotes),
- [5] "This is the message.\n" (i.e. the text of the here document),
- [6] "EOMSG" (i.e. the right delimiter of the here document),
- [7..10] "" (a here document has no second left delimiter, second text, second right delimiter, or trailing modifiers).

However, the matching position of the input variable would be set to "exit;" (i.e. *after* the closing delimiter of the here document), which would cause the earlier " || die;\nexit;" to be skipped in any sequence of code fragment extractions.

To avoid this problem, when it encounters a here document while extracting from a modifiable string, `extract_quotelike` silently rearranges the string to an equivalent piece of Perl:

```
<<'EOMSG'
This is the message.
EOMSG
|| die;
exit;
```

in which the here document *is* contiguous. It still leaves the matching position after the here document, but now the rest of the line on which the here document starts is not skipped.

To prevent `<extract_quotelike` from mucking about with the input in this way (this is the only case where a list-context `extract_quotelike` does so), you can pass the input variable as an interpolated literal:

```
$quotelike = extract_quotelike("$var");
```

**extract\_codeblock**

`extract_codeblock` attempts to recognize and extract a balanced bracket delimited substring that may contain unbalanced brackets inside Perl quotes or quotelike operations. That is, `extract_codeblock` is like a combination of "`extract_bracketed`" and "`extract_quotelike`".



`extract_codeblock` takes the same initial three parameters as `extract_bracketed`: a text to process, a set of delimiter brackets to look for, and a prefix to match first. It also takes an optional fourth parameter, which allows the outermost delimiter brackets to be specified separately (see below).

Omitting the first argument (input text) means process `$_` instead. Omitting the second argument (delimiter brackets) indicates that only `{ }` is to be used. Omitting the third argument (prefix argument) implies optional whitespace at the start. Omitting the fourth argument (outermost delimiter brackets) indicates that the value of the second argument is to be used for the outermost delimiters.

Once the prefix and the outermost opening delimiter bracket have been recognized, code blocks are extracted by stepping through the input text and trying the following alternatives in sequence:

1. Try and match a closing delimiter bracket. If the bracket was the same species as the last opening bracket, return the substring to that point. If the bracket was mismatched, return an error.
2. Try to match a quote or quotelike operator. If found, call `extract_quotelike` to eat it. If `extract_quotelike` fails, return the error it returned. Otherwise go back to step 1.
3. Try to match an opening delimiter bracket. If found, call `extract_codeblock` recursively to eat the embedded block. If the recursive call fails, return an error. Otherwise, go back to step 1.
4. Unconditionally match a bareword or any other single character, and then go back to step 1.

Examples:

```
Find a while loop in the text
 if ($text =~ s/. *?while\s*\{ /\{ /)
 {
 $loop = "while " . extract_codeblock($text);
 }

Remove the first round-bracketed list (which may include
round- or curly-bracketed code blocks or quotelike operators)
 extract_codeblock $text, "(){}", '[^()*]*';
```

The ability to specify a different outermost delimiter bracket is useful in some circumstances. For example, in the `Parse::RecDescent` module, parser actions which are to be performed only on a successful parse are specified using a `<defer: ...>` directive. For example:

```
sentence: subject verb object
 <defer: { $::theVerb = $item{verb} } >
```

`Parse::RecDescent` uses `extract_codeblock($text, '{ }<>')` to extract the code within the `<defer: ...>` directive, but there's a problem.

A deferred action like this:

```
<defer: { if ($count > 10) { $count -- } } >
```

will be incorrectly parsed as:

```
<defer: { if ($count >
```

because the "less than" operator is interpreted as a closing delimiter.

But, by extracting the directive using `extract_codeblock($text, '{ }', undef, '<>')` the `>` character is only treated as a delimited at the outermost level of the code block, so the directive is parsed correctly.

### **extract\_multiple**

The `extract_multiple` subroutine takes a string to be processed and a list of extractors (subroutines or regular expressions) to apply to that string.

In an array context `extract_multiple` returns an array of substrings of the original string, as extracted by the specified extractors. In a scalar context, `extract_multiple` returns the first substring successfully extracted from the original string. In both scalar and void contexts the original string has the first successfully extracted substring removed from it. In all contexts `extract_multiple` starts at the current `pos` of the string, and sets that `pos` appropriately after it matches.

Hence, the aim of a call to `extract_multiple` in a list context is to split the processed string into as many non-overlapping fields as possible, by repeatedly applying each of the specified extractors to the remainder of the string. Thus `extract_multiple` is a generalized form of Perl's `split` subroutine.

The subroutine takes up to four optional arguments:

1. A string to be processed (`$_` if the string is omitted or `undef`)
2. A reference to a list of subroutine references and/or `qr//` objects and/or literal strings and/or hash references, specifying the extractors to be used to split the string. If this argument is omitted (or `undef`) the list:

```
[
 sub { extract_variable($_[0], '') },
 sub { extract_quotelike($_[0], '') },
 sub { extract_codeblock($_[0], '{', '}', '') },
]
```

is used.

3. A number specifying the maximum number of fields to return. If this argument is omitted (or `undef`), `split` continues as long as possible.

If the third argument is *N*, then extraction continues until *N* fields have been successfully extracted, or until the string has been completely processed.

Note that in scalar and void contexts the value of this argument is automatically reset to 1 (under `-w`, a warning is issued if the argument has to be reset).

4. A value indicating whether unmatched substrings (see below) within the text should be skipped or returned as fields. If the value is true, such substrings are skipped. Otherwise, they are returned.

The extraction process works by applying each extractor in sequence to the text string.

If the extractor is a subroutine it is called in a list context and is expected to return a list of a single element, namely the extracted text. It may optionally also return two further arguments: a string representing the text left after extraction (like `$'` for a pattern match), and a string representing any prefix skipped before the extraction (like `$`` in a pattern match). Note that this is designed to facilitate the use of other `Text::Balanced` subroutines with `extract_multiple`. Note too that the value returned by an extractor subroutine need not bear any relationship to the corresponding substring of the original text (see examples below).

If the extractor is a precompiled regular expression or a string, it is matched against the text in a scalar context with a leading `'G'` and the `gc` modifiers enabled. The extracted value is either `$1` if that variable is defined after the match, or else the complete match (i.e. `$&`).

If the extractor is a hash reference, it must contain exactly one element. The value of that element is one of the above extractor types (subroutine reference, regular expression, or string). The key of that element is the name of a class into which the successful return value of the extractor will be blessed.

If an extractor returns a defined value, that value is immediately treated as the next extracted field and pushed onto the list of fields. If the extractor was specified in a hash reference, the field is also blessed into the appropriate class,

If the extractor fails to match (in the case of a regex extractor), or returns an empty list or an undefined value (in the case of a subroutine extractor), it is assumed to have failed to extract. If none of the extractor subroutines succeeds, then one character is extracted from the start of the text and the extraction subroutines

reapplied. Characters which are thus removed are accumulated and eventually become the next field (unless the fourth argument is true, in which case they are discarded).

For example, the following extracts substrings that are valid Perl variables:

```
@fields = extract_multiple($text,
 [sub { extract_variable($_[0]) }],
 undef, 1);
```

This example separates a text into fields which are quote delimited, curly bracketed, and anything else. The delimited and bracketed parts are also blessed to identify them (the "anything else" is unblessed):

```
@fields = extract_multiple($text,
 [
 { Delim => sub { extract_delimited($_[0], q{'"'}) } },
 { Brack => sub { extract_bracketed($_[0], '{ }') } },
]);
```

This call extracts the next single substring that is a valid Perl quotelike operator (and removes it from \$text):

```
$quotelike = extract_multiple($text,
 [
 sub { extract_quotelike($_[0]) },
], undef, 1);
```

Finally, here is yet another way to do comma-separated value parsing:

```
@fields = extract_multiple($csv_text,
 [
 sub { extract_delimited($_[0], q{'"'}) },
 qr/([^\,]+)(.*)/,
],
 undef, 1);
```

The list in the second argument means: *"Try and extract a ' or " delimited string, otherwise extract anything up to a comma..."*. The undef third argument means: *"...as many times as possible..."*, and the true value in the fourth argument means *"...discarding anything else that appears (i.e. the commas)"*.

If you wanted the commas preserved as separate fields (i.e. like split does if your split pattern has capturing parentheses), you would just make the last parameter undefined (or remove it).

### gen\_delimited\_pat

The `gen_delimited_pat` subroutine takes a single (string) argument and builds a Friedl-style optimized regex that matches a string delimited by any one of the characters in the single argument. For example:

```
gen_delimited_pat(q{'"'})
```

returns the regex:

```
(?:\"(?:\\\"|(?!\")\S)*\"|\'(?:\\\'|(?!\\')\S)*\')
```

Note that the specified delimiters are automatically quotemeta'd.

A typical use of `gen_delimited_pat` would be to build special purpose tags for `extract_tagged`. For example, to properly ignore "empty" XML elements (which might contain quoted strings):

```
my $empty_tag = '<(' . gen_delimited_pat(q{'"'}) . ' |.)+>';
extract_tagged($text, undef, undef, undef, {ignore => [$empty_tag]});
```

`gen_delimited_pat` may also be called with an optional second argument, which specifies the "escape" character(s) to be used for each delimiter. For example to match a Pascal-style string (where ' is the

delimiter and `` is a literal ` within the string):

```
gen_delimited_pat(q{''},q{''}) ;
```

Different escape characters can be specified for different delimiters. For example, to specify that `/` is the escape for single quotes and `%` is the escape for double quotes:

```
gen_delimited_pat(q{' "%'},q{'/%'}) ;
```

If more delimiters than escape chars are specified, the last escape char is used for the remaining delimiters. If no escape char is specified for a given specified delimiter, `` is used.

Note that `gen_delimited_pat` was previously called `delimited_pat`. That name may still be used, but is now deprecated.

## DIAGNOSTICS

In a list context, all the functions return `(undef,$original_text)` on failure. In a scalar context, failure is indicated by returning `undef` (in this case the input text is not modified in any way).

In addition, on failure in *any* context, the `$@` variable is set. Accessing `$@->{error}` returns one of the error diagnostics listed below. Accessing `$@->{pos}` returns the offset into the original string at which the error was detected (although not necessarily where it occurred!) Printing `$@` directly produces the error message, with the offset appended. On success, the `$@` variable is guaranteed to be `undef`.

The available diagnostics are:

Did not find a suitable bracket: "%s"

The delimiter provided to `extract_bracketed` was not one of `() [] <> {}`.

Did not find prefix: /%s/

A non-optional prefix was specified but wasn't found at the start of the text.

Did not find opening bracket after prefix: "%s"

`extract_bracketed` or `extract_codeblock` was expecting a particular kind of bracket at the start of the text, and didn't find it.

No quotelike operator found after prefix: "%s"

`extract_quotelike` didn't find one of the quotelike operators `q`, `qq`, `qw`, `qx`, `s`, `tr` or `y` at the start of the substring it was extracting.

Unmatched closing bracket: "%c"

`extract_bracketed`, `extract_quotelike` or `extract_codeblock` encountered a closing bracket where none was expected.

Unmatched opening bracket(s): "%s"

`extract_bracketed`, `extract_quotelike` or `extract_codeblock` ran out of characters in the text before closing one or more levels of nested brackets.

Unmatched embedded quote (%s)

`extract_bracketed` attempted to match an embedded quoted substring, but failed to find a closing quote to match it.

Did not find closing delimiter to match '%s'

`extract_quotelike` was unable to find a closing delimiter to match the one that opened the quote-like operation.

Mismatched closing bracket: expected "%c" but found "%s"

`extract_bracketed`, `extract_quotelike` or `extract_codeblock` found a valid bracket delimiter, but it was the wrong species. This usually indicates a nesting error, but may indicate incorrect quoting or escaping.

**No block delimiter found after quotelike "%s"**

`extract_quotelike` or `extract_codeblock` found one of the quotelike operators `q`, `qq`, `qw`, `qx`, `s`, `tr` or `y` without a suitable block after it.

**Did not find leading dereferencer**

`extract_variable` was expecting one of `'$'`, `'@'`, or `'%'` at the start of a variable, but didn't find any of them.

**Bad identifier after dereferencer**

`extract_variable` found a `'$'`, `'@'`, or `'%'` indicating a variable, but that character was not followed by a legal Perl identifier.

**Did not find expected opening bracket at %s**

`extract_codeblock` failed to find any of the outermost opening brackets that were specified.

**Improperly nested codeblock at %s**

A nested code block was found that started with a delimiter that was specified as being only to be used as an outermost bracket.

**Missing second block for quotelike "%s"**

`extract_codeblock` or `extract_quotelike` found one of the quotelike operators `s`, `tr` or `y` followed by only one block.

**No match found for opening bracket**

`extract_codeblock` failed to find a closing bracket to match the outermost opening bracket.

**Did not find opening tag: %s/**

`extract_tagged` did not find a suitable opening tag (after any specified prefix was removed).

**Unable to construct closing tag to match: %s/**

`extract_tagged` matched the specified opening tag and tried to modify the matched text to produce a matching closing tag (because none was specified). It failed to generate the closing tag, almost certainly because the opening tag did not start with a bracket of some kind.

**Found invalid nested tag: %s**

`extract_tagged` found a nested tag that appeared in the "reject" list (and the failure mode was not "MAX" or "PARA").

**Found unbalanced nested tag: %s**

`extract_tagged` found a nested opening tag that was not matched by a corresponding nested closing tag (and the failure mode was not "MAX" or "PARA").

**Did not find closing tag**

`extract_tagged` reached the end of the text without finding a closing tag to match the original opening tag (and the failure mode was not "MAX" or "PARA").

**AUTHOR**

Damian Conway ([damian@conway.org](mailto:damian@conway.org))

**BUGS AND IRRITATIONS**

There are undoubtedly serious bugs lurking somewhere in this code, if only because parts of it give the impression of understanding a great deal more about Perl than they really do.

Bug reports and other feedback are most welcome.

**COPYRIGHT**

Copyright (c) 1997–2001, Damian Conway. All Rights Reserved.  
This module is free software. It may be used, redistributed  
and/or modified under the same terms as Perl itself.

**NAME**

Text::ParseWords – parse text into an array of tokens or array of arrays

**SYNOPSIS**

```
use Text::ParseWords;
@lists = &nested_quotewords($delim, $keep, @lines);
@words = "ewords($delim, $keep, @lines);
@words = &shellwords(@lines);
@words = &parse_line($delim, $keep, $line);
@words = &old_shellwords(@lines); # DEPRECATED!
```

**DESCRIPTION**

The `&nested_quotewords()` and `&quotewords()` functions accept a delimiter (which can be a regular expression) and a list of lines and then breaks those lines up into a list of words ignoring delimiters that appear inside quotes. `&quotewords()` returns all of the tokens in a single long list, while `&nested_quotewords()` returns a list of token lists corresponding to the elements of `@lines`. `&parse_line()` does tokenizing on a single string. The `&*quotewords()` functions simply call `&parse_lines()`, so if you're only splitting one line you can call `&parse_lines()` directly and save a function call.

The `$keep` argument is a boolean flag. If true, then the tokens are split on the specified delimiter, but all other characters (quotes, backslashes, etc.) are kept in the tokens. If `$keep` is false then the `&*quotewords()` functions remove all quotes and backslashes that are not themselves backslash-escaped or inside of single quotes (i.e., `&quotewords()` tries to interpret these characters just like the Bourne shell). NB: these semantics are significantly different from the original version of this module shipped with Perl 5.000 through 5.004. As an additional feature, `$keep` may be the keyword "delimiters" which causes the functions to preserve the delimiters in each string as tokens in the token lists, in addition to preserving quote and backslash characters.

`&shellwords()` is written as a special case of `&quotewords()`, and it does token parsing with whitespace as a delimiter— similar to most Unix shells.

**EXAMPLES**

The sample program:

```
use Text::ParseWords;
@words = "ewords('\s+', 0, q{this is "a test" of\ quotewords \"for you});
$i = 0;
foreach (@words) {
 print "$i: <$_>\n";
 $i++;
}
```

produces:

```
0: <this>
1: <is>
2: <a test>
3: <of quotewords>
4: <"for>
5: <you>
```

demonstrating:

- 0 a simple word
- 1 multiple spaces are skipped because of our `$delim`

- 2 use of quotes to include a space in a word
- 3 use of a backslash to include a space in a word
- 4 use of a backslash to remove the special meaning of a double-quote
- 5 another simple word (note the lack of effect of the backslashed double-quote)

Replacing `&quotewords('\s+', 0, q{this is...})` with `&shellwords(q{this is...})` is a simpler way to accomplish the same thing.

## AUTHORS

Maintainer is Hal Pomeranz <pomeranz@netcom.com, 1994–1997 (Original author unknown). Much of the code for `&parse_line()` (including the primary regexp) from Joerk Behrends <jbehrends@multimediaproduzenten.de.

Examples section another documentation provided by John Heidemann <johnh@ISI.EDU

Bug reports, patches, and nagging provided by lots of folks— thanks everybody! Special thanks to Michael Schwern <schwern@envirolink.org for assuring me that a `&nested_quotewords()` would be useful, and to Jeff Friedl <jfriedl@yahoo-inc.com for telling me not to worry about error-checking (sort of— you had to be there).

## NAME

Text::Soundex – Implementation of the Soundex Algorithm as Described by Knuth

## SYNOPSIS

```
use Text::Soundex;

$code = soundex $string; # get soundex code for a string
@codes = soundex @list; # get list of codes for list of strings

set value to be returned for strings without soundex code

$soundex_nocode = 'Z000';
```

## DESCRIPTION

This module implements the soundex algorithm as described by Donald Knuth in Volume 3 of **The Art of Computer Programming**. The algorithm is intended to hash words (in particular surnames) into a small space using a simple model which approximates the sound of the word when spoken by an English speaker. Each word is reduced to a four character string, the first character being an upper case letter and the remaining three being digits.

If there is no soundex code representation for a string then the value of `$soundex_nocode` is returned. This is initially set to `undef`, but many people seem to prefer an *unlikely* value like `Z000` (how unlikely this is depends on the data set being dealt with.) Any value can be assigned to `$soundex_nocode`.

In scalar context `soundex` returns the soundex code of its first argument, and in list context a list is returned in which each element is the soundex code for the corresponding argument passed to `soundex` e.g.

```
@codes = soundex qw(Mike Stok);
```

leaves `@codes` containing `('M200', 'S320')`.

## EXAMPLES

Knuth's examples of various names and the soundex codes they map to are listed below:

```
Euler, Ellery -> E460
Gauss, Ghosh -> G200
Hilbert, Heilbronn -> H416
Knuth, Kant -> K530
Lloyd, Ladd -> L300
Lukasiewicz, Lissajous -> L222
```

so:

```
$code = soundex 'Knuth'; # $code contains 'K530'
@list = soundex qw(Lloyd Gauss); # @list contains 'L300', 'G200'
```

## LIMITATIONS

As the soundex algorithm was originally used a **long** time ago in the US it considers only the English alphabet and pronunciation.

As it is mapping a large space (arbitrary length strings) onto a small space (single letter plus 3 digits) no inference can be made about the similarity of two strings which end up with the same soundex code. For example, both Hilbert and Heilbronn end up with a soundex code of H416.

## AUTHOR

This code was implemented by Mike Stok ([stok@cybercom.net](mailto:stok@cybercom.net)) from the description given by Knuth. Ian Phillipps ([ian@pipex.net](mailto:ian@pipex.net)) and Rich Pinder ([rpinder@hsc.usc.edu](mailto:rpinder@hsc.usc.edu)) supplied ideas and spotted mistakes.



**NAME**

Text::Tabs — expand and unexpand tabs per the unix `expand(1)` and `unexpand(1)`

**SYNOPSIS**

```
use Text::Tabs;

$tabstop = 4;
@lines_without_tabs = expand(@lines_with_tabs);
@lines_with_tabs = unexpand(@lines_without_tabs);
```

**DESCRIPTION**

Text::Tabs does about what the unix utilities `expand(1)` and `unexpand(1)` do. Given a line with tabs in it, `expand` will replace the tabs with the appropriate number of spaces. Given a line with or without tabs in it, `unexpand` will add tabs when it can save bytes by doing so. Invisible compression with plain ascii!

**BUGS**

`expand` doesn't handle newlines very quickly — do not feed it an entire document in one string. Instead feed it an array of lines.

**AUTHOR**

David Muir Sharnoff <muir@idiom.com>

**NAME**

Text::Wrap – line wrapping to form simple paragraphs

**SYNOPSIS****Example 1**

```
use Text::Wrap

$initial_tab = "\t"; # Tab before first line
$subsequent_tab = ""; # All other lines flush left

print wrap($initial_tab, $subsequent_tab, @text);
print fill($initial_tab, $subsequent_tab, @text);

@lines = wrap($initial_tab, $subsequent_tab, @text);

@paragraphs = fill($initial_tab, $subsequent_tab, @text);
```

**Example 2**

```
use Text::Wrap qw(wrap $columns $huge);

$columns = 132; # Wrap at 132 characters
$huge = 'die';
$huge = 'wrap';
$huge = 'overflow';
```

**Example 3**

```
use Text::Wrap

$Text::Wrap::columns = 72;
print wrap(' ', ' ', @text);
```

**DESCRIPTION**

`Text::Wrap::wrap()` is a very simple paragraph formatter. It formats a single paragraph at a time by breaking lines at word boundaries. Indentation is controlled for the first line (`$initial_tab`) and all subsequent lines (`$subsequent_tab`) independently. Please note: `$initial_tab` and `$subsequent_tab` are the literal strings that will be used: it is unlikely you would want to pass in a number.

`Text::Wrap::fill()` is a simple multi-paragraph formatter. It formats each paragraph separately and then joins them together when it's done. It will destroy any whitespace in the original text. It breaks text into paragraphs by looking for whitespace after a newline. In other respects it acts like `wrap()`.

**OVERRIDES**

`Text::Wrap::wrap()` has a number of variables that control its behavior. Because other modules might be using `Text::Wrap::wrap()` it is suggested that you leave these variables alone! If you can't do that, then use `local($Text::Wrap::VARIABLE) = YOURVALUE` when you change the values so that the original value is restored. This `local()` trick will not work if you import the variable into your own namespace.

Lines are wrapped at `$Text::Wrap::columns` columns. `$Text::Wrap::columns` should be set to the full width of your output device. In fact, every resulting line will have length of no more than `$columns - 1`.

It is possible to control which characters terminate words by modifying `$Text::Wrap::break`. Set this to a string such as `'[\s:]'` (to break before spaces or colons) or a pre-compiled regexp such as `qr/[\s']/` (to break before spaces or apostrophes). The default is simply `'\s'`; that is, words are terminated by spaces. (This means, among other things, that trailing punctuation such as full stops or commas stay with the word they are "attached" to.)

Beginner note: In example 2, above `$columns` is imported into the local namespace, and set locally. In example 3, `$Text::Wrap::columns` is set in its own namespace without importing it.

`Text::Wrap::wrap()` starts its work by expanding all the tabs in its input into spaces. The last thing it does is to turn spaces back into tabs. If you do not want tabs in your results, set

`$Text::Wrap::unexpand` to a false value. Likewise if you do not want to use 8-character tabstops, set `$Text::Wrap::tabstop` to the number of characters you do want for your tabstops.

If you want to separate your lines with something other than `\n` then set `$Text::Wrap::separator` to your preference.

When words that are longer than `$columns` are encountered, they are broken up. `wrap()` adds a `"\n"` at column `$columns`. This behavior can be overridden by setting `$huge` to `'die'` or to `'overflow'`. When set to `'die'`, large words will cause `die()` to be called. When set to `'overflow'`, large words will be left intact.

Historical notes: `'die'` used to be the default value of `$huge`. Now, `'wrap'` is the default value.

## EXAMPLE

```
print wrap("\t","", "This is a bit of text that forms
a normal book-style paragraph");
```

## AUTHOR

David Muir Sharnoff <muir@idiom.com> with help from Tim Pierce and many many others.

**NAME**

Thread – manipulate threads in Perl (for old code only)

**CAVEAT**

Perl has two thread models.

In Perl 5.005 the thread model was that all data is implicitly shared and shared access to data has to be explicitly synchronized. This model is called "5005threads".

In Perl 5.6 a new model was introduced in which all is was thread local and shared access to data has to be explicitly declared. This model is called "ithreads", for "interpreter threads".

In Perl 5.6 the ithreads model was not available as a public API, only as an internal API that was available for extension writers, and to implement `fork()` emulation on Win32 platforms.

In Perl 5.8 the ithreads model became available through the `threads` module.

Neither model is configured by default into Perl (except, as mentioned above, in Win32 ithreads are always available.) You can see your Perl's threading configuration by running `perl -V` and looking for the *use...threads* variables, or inside script by `use Config;` and testing for `$Config{use5005threads}` and `$Config{useithreads}`.

For old code and interim backwards compatibility, the Thread module has been reworked to function as a frontend for both 5005threads and ithreads.

Note that the compatibility is not complete: because the data sharing models are directly opposed, anything to do with data sharing has to be thought differently. With the ithreads you must explicitly `share()` variables between the threads.

For new code the use of the Thread module is discouraged and the direct use of the `threads` and `threads::shared` modules is encouraged instead.

Finally, note that there are many known serious problems with the 5005threads, one of the least of which is that regular expression match variables like `$1` are not threadsafe, that is, they easily get corrupted by competing threads. Other problems include more insidious data corruption and mysterious crashes. You are seriously urged to use ithreads instead.

**SYNOPSIS**

```
use Thread;

my $t = Thread->new(&start_sub, @start_args);

$result = $t->join;
$result = $t->eval;
$t->detach;

if ($t->done) {
 $t->join;
}

if ($t->equal($another_thread)) {
 # ...
}

yield();

my $tid = Thread->self->tid;

lock($scalar);
lock(@array);
lock(%hash);
```

```
lock(\&sub); # not available with ithreads

$flags = $t->flags; # not available with ithreads

my @list = Thread->list; # not available with ithreads

use Thread 'async';
```

## DESCRIPTION

The Thread module provides multithreading support for perl.

## FUNCTIONS

```
$thread = Thread->new(\&start_sub)
$thread = Thread->new(\&start_sub, LIST)
```

`new` starts a new thread of execution in the referenced subroutine. The optional list is passed as parameters to the subroutine. Execution continues in both the subroutine and the code after the new call.

`Thread->new` returns a thread object representing the newly created thread.

### lock VARIABLE

`lock` places a lock on a variable until the lock goes out of scope.

If the variable is locked by another thread, the `lock` call will block until it's available. `lock` is recursive, so multiple calls to `lock` are safe—the variable will remain locked until the outermost lock on the variable goes out of scope.

Locks on variables only affect `lock` calls—they do *not* affect normal access to a variable. (Locks on subs are different, and covered in a bit.) If you really, *really* want locks to block access, then go ahead and tie them to something and manage this yourself. This is done on purpose. While managing access to variables is a good thing, Perl doesn't force you out of its living room...

If a container object, such as a hash or array, is locked, all the elements of that container are not locked. For example, if a thread does a `lock @a`, any other thread doing a `lock ($a[12])` won't block.

With 5005threads you may also lock a sub, using `lock &sub`. Any calls to that sub from another thread will block until the lock is released. This behaviour is not equivalent to declaring the sub with the `locked` attribute. The `locked` attribute serializes access to a subroutine, but allows different threads non-simultaneous access. `lock &sub`, on the other hand, will not allow *any* other thread access for the duration of the lock.

Finally, `lock` will traverse up references exactly *one* level. `lock(\$a)` is equivalent to `lock($a)`, while `lock(\(\$a)` is not.

### async BLOCK;

`async` creates a thread to execute the block immediately following it. This block is treated as an anonymous sub, and so must have a semi-colon after the closing brace. Like `Thread->new`, `async` returns a thread object.

### Thread->self

The `Thread->self` function returns a thread object that represents the thread making the `Thread->self` call.

### cond\_wait VARIABLE

The `cond_wait` function takes a **locked** variable as a parameter, unlocks the variable, and blocks until another thread does a `cond_signal` or `cond_broadcast` for that same locked variable. The variable that `cond_wait` blocked on is relocked after the `cond_wait` is satisfied. If there are multiple threads `cond_waiting` on the same variable, all but one will

reblock waiting to reacquire the lock on the variable. (So if you're only using `cond_wait` for synchronization, give up the lock as soon as possible.)

#### `cond_signal` VARIABLE

The `cond_signal` function takes a locked variable as a parameter and unblocks one thread that's `cond_waiting` on that variable. If more than one thread is blocked in a `cond_wait` on that variable, only one (and which one is indeterminate) will be unblocked.

If there are no threads blocked in a `cond_wait` on the variable, the signal is discarded.

#### `cond_broadcast` VARIABLE

The `cond_broadcast` function works similarly to `cond_signal`. `cond_broadcast`, though, will unblock **all** the threads that are blocked in a `cond_wait` on the locked variable, rather than only one.

`yield` The `yield` function allows another thread to take control of the CPU. The exact results are implementation-dependent.

### METHODS

`join` `join` waits for a thread to end and returns any values the thread exited with. `join` will block until the thread has ended, though it won't block if the thread has already terminated.

If the thread being joined died, the error it died with will be returned at this time. If you don't want the thread performing the `join` to die as well, you should either wrap the `join` in an `eval` or use the `eval` thread method instead of `join`.

`eval` The `eval` method wraps an `eval` around a `join`, and so waits for a thread to exit, passing along any values the thread might have returned. Errors, of course, get placed into `$@`. (Not available with `ithreads`.)

`detach` `detach` tells a thread that it is never going to be joined i.e. that all traces of its existence can be removed once it stops running. Errors in detached threads will not be visible anywhere – if you want to catch them, you should use `$SIG{__DIE__}` or something like that.

`equal` `equal` tests whether two thread objects represent the same thread and returns true if they do.

`tid` The `tid` method returns the `tid` of a thread. The `tid` is a monotonically increasing integer assigned when a thread is created. The main thread of a program will have a `tid` of zero, while subsequent threads will have `tids` assigned starting with one.

`flags` The `flags` method returns the flags for the thread. This is the integer value corresponding to the internal flags for the thread, and the value may not be all that meaningful to you. (Not available with `ithreads`.)

`done` The `done` method returns true if the thread you're checking has finished, and false otherwise. (Not available with `ithreads`.)

### LIMITATIONS

The sequence number used to assign `tids` is a simple integer, and no checking is done to make sure the `tid` isn't currently in use. If a program creates more than  $2^{32} - 1$  threads in a single run, threads may be assigned duplicate `tids`. This limitation may be lifted in a future version of Perl.

### SEE ALSO

[\*threads::shared\*](#) (not available with `5005threads`)

[\*attributes\*](#), [\*Thread::Queue\*](#), [\*Thread::Semaphore\*](#), [\*Thread::Specific\*](#) (not available with `ithreads`)

**NAME**

Thread::Queue – thread-safe queues

**SYNOPSIS**

```
use Thread::Queue;
my $q = new Thread::Queue;
$q->enqueue("foo", "bar");
my $foo = $q->dequeue; # The "bar" is still in the queue.
my $foo = $q->dequeue_nb; # returns "bar", or undef if the queue was empty
my $left = $q->pending; # returns the number of items still in the queue
```

**DESCRIPTION**

A queue, as implemented by Thread::Queue is a thread-safe data structure much like a list. Any number of threads can safely add elements to the end of the list, or remove elements from the head of the list. (Queues don't permit adding or removing elements from the middle of the list).

**FUNCTIONS AND METHODS**

**new**        The new function creates a new empty queue.

**enqueue LIST**

The enqueue method adds a list of scalars on to the end of the queue. The queue will grow as needed to accommodate the list.

**dequeue**    The dequeue method removes a scalar from the head of the queue and returns it. If the queue is currently empty, dequeue will block the thread until another thread enqueues a scalar.

**dequeue\_nb**

The dequeue\_nb method, like the dequeue method, removes a scalar from the head of the queue and returns it. Unlike dequeue, though, dequeue\_nb won't block if the queue is empty, instead returning undef.

**pending**    The pending method returns the number of items still in the queue.

**SEE ALSO**

*threads, threads::shared*

**NAME**

Thread::Semaphore – thread-safe semaphores

**SYNOPSIS**

```
use Thread::Semaphore;
my $s = new Thread::Semaphore;
$s->up; # Also known as the semaphore V -operation.
The guarded section is here
$s->down; # Also known as the semaphore P -operation.

The default semaphore value is 1.
my $s = new Thread::Semaphore($initial_value);
$s->up($up_value);
$s->down($down_value);
```

**DESCRIPTION**

Semaphores provide a mechanism to regulate access to resources. Semaphores, unlike locks, aren't tied to particular scalars, and so may be used to control access to anything you care to use them for.

Semaphores don't limit their values to zero or one, so they can be used to control access to some resource that there may be more than one of. (For example, filehandles). Increment and decrement amounts aren't fixed at one either, so threads can reserve or return multiple resources at once.

**FUNCTIONS AND METHODS**

**new**  
**new NUMBER**

**new** creates a new semaphore, and initializes its count to the passed number. If no number is passed, the semaphore's count is set to one.

**down**  
**down NUMBER**

The **down** method decreases the semaphore's count by the specified number, or by one if no number has been specified. If the semaphore's count would drop below zero, this method will block until such time that the semaphore's count is equal to or larger than the amount you're downing the semaphore's count by.

**up**  
**up NUMBER**

The **up** method increases the semaphore's count by the number specified, or by one if no number has been specified. This will unblock any thread blocked trying to down the semaphore if the up raises the semaphore count above the amount that the downs are trying to decrement it by.



**NAME**

Thread::Signal – Start a thread which runs signal handlers reliably (for old code)

**CAVEAT**

For new code the use of the `Thread::Signal` module is discouraged and the direct use of the `threads` and associated modules is encouraged instead.

However, there is no direct equivalent of the `Thread::Signal` module in the new implementation of threads. On the bright side: signals are now delivered reliably to Perl programs that do not use threads. The handling of signals with the new threading features is up to the underlying thread implementation that is being used and may therefore be less reliable.

If you want to specify a thread-specific signal, you can alter the `%SIG` hash in the thread where you want to handle a signal differently from other threads. This at least seems to work under Linux. But there are no guarantees and your mileage may vary.

For the whole story about the development of threads in Perl, and why you should **not** be using this module unless you know what you're doing, see the CAVEAT of the `Thread` module.

**SYNOPSIS**

```
use Thread::Signal;

$SIG{HUP} = \&some_handler;
```

**DESCRIPTION**

The `Thread::Signal` module starts up a special signal handler thread. All signals to the process are delivered to it and it runs the associated `$SIG{FOO}` handlers for them. Without this module, signals arriving at inopportune moments (such as when perl's internals are in the middle of updating critical structures) cause the perl code of the handler to be run unsafely which can cause memory corruption or worse.

**BUGS**

This module changes the semantics of signal handling slightly in that the signal handler is run separately from the main thread (and in parallel with it). This means that tricks such as calling `die` from a signal handler behave differently (and, in particular, can't be used to exit directly from a system call).

**NAME**

Thread::Specific – thread-specific keys

**SYNOPSIS**

```
use Thread::Specific;
my $k = key_create Thread::Specific;
```

**DESCRIPTION**

`key_create` returns a unique thread-specific key.

**NAME**

threads::shared – Perl extension for sharing data structures between threads

**SYNOPSIS**

```
use threads;
use threads::shared;

my $var : shared;

my($scalar, @array, %hash);
share($scalar);
share(@array);
share(%hash);
my $bar = &share([]);
$hash{bar} = &share({});

{ lock(%hash); ... }

cond_wait($scalar);
cond_broadcast(@array);
cond_signal(%hash);
```

**DESCRIPTION**

By default, variables are private to each thread, and each newly created thread gets a private copy of each existing variable. This module allows you to share variables across different threads (and pseudoforks on Win32). It is used together with the threads module.

**EXPORT**

share, lock, cond\_wait, cond\_signal, cond\_broadcast

Note that if this module is imported when threads has not yet been loaded, then these functions all become no-ops. This makes it possible to write modules that will work in both threaded and non-threaded environments.

**FUNCTIONS****share VARIABLE**

share takes a value and marks it as shared. You can share a scalar, array, hash, scalar ref, array ref or hash ref. share will return the shared rvalue.

share will traverse up references exactly *one* level. share(\\$a) is equivalent to share(\$a), while share(\\\$a) is not.

A variable can also be marked as shared at compile time by using the shared attribute: my \$var : shared.

If you want to share a newly created reference unfortunately you need to use &share([]) and &share({}) syntax due to problems with Perl's prototyping.

**lock VARIABLE**

lock places a lock on a variable until the lock goes out of scope. If the variable is locked by another thread, the lock call will block until it's available. lock is recursive, so multiple calls to lock are safe — the variable will remain locked until the outermost lock on the variable goes out of scope.

If a container object, such as a hash or array, is locked, all the elements of that container are not locked. For example, if a thread does a lock @a, any other thread doing a lock(\$a[12]) won't block.

lock will traverse up references exactly *one* level. lock(\\$a) is equivalent to lock(\$a), while lock(\\\$a) is not.

Note that you cannot explicitly unlock a variable; you can only wait for the lock to go out of scope. If you need more fine-grained control, see [Thread::Semaphore](#).

#### cond\_wait VARIABLE

The `cond_wait` function takes a **locked** variable as a parameter, unlocks the variable, and blocks until another thread does a `cond_signal` or `cond_broadcast` for that same locked variable. The variable that `cond_wait` blocked on is relocked after the `cond_wait` is satisfied. If there are multiple threads `cond_waiting` on the same variable, all but one will reblock waiting to reacquire the lock on the variable. (So if you're only using `cond_wait` for synchronisation, give up the lock as soon as possible). The two actions of unlocking the variable and entering the blocked wait state are atomic. The two actions of exiting from the blocked wait state and relocking the variable are not.

It is important to note that the variable can be notified even if no thread `cond_signal` or `cond_broadcast` on the variable. It is therefore important to check the value of the variable and go back to waiting if the requirement is not fulfilled.

#### cond\_signal VARIABLE

The `cond_signal` function takes a **locked** variable as a parameter and unblocks one thread that's `cond_waiting` on that variable. If more than one thread is blocked in a `cond_wait` on that variable, only one (and which one is indeterminate) will be unblocked.

If there are no threads blocked in a `cond_wait` on the variable, the signal is discarded. By always locking before signaling, you can (with care), avoid signaling before another thread has entered `cond_wait()`.

`cond_signal` will normally generate a warning if you attempt to use it on an unlocked variable. On the rare occasions where doing this may be sensible, you can skip the warning with

```
{ no warnings 'threads'; cond_signal($foo) }
```

#### cond\_broadcast VARIABLE

The `cond_broadcast` function works similarly to `cond_signal`. `cond_broadcast`, though, will unblock **all** the threads that are blocked in a `cond_wait` on the locked variable, rather than only one.

## NOTES

`threads::shared` is designed to disable itself silently if threads are not available. If you want access to threads, you must use `threads` before you use `threads::shared`. `threads` will emit a warning if you use it after `threads::shared`.

## BUGS

`bless` is not supported on shared references. In the current version, `bless` will only bless the thread local reference and the blessing will not propagate to the other threads. This is expected to be implemented in a future version of Perl.

Does not support splice on arrays!

Taking references to the elements of shared arrays and hashes does not autovivify the elements, and neither does slicing a shared array/hash over non-existent indices/keys autovivify the elements.

`share()` allows you to share `$hashref->{key}` without giving any error message. But the `$hashref->{key}` is **not** shared, causing the error "locking can only be used on shared values" to occur when you attempt to lock `$hashref->{key}`.

## AUTHOR

Arthur Bergman <arthur at contiller.se>

`threads::shared` is released under the same license as Perl

Documentation borrowed from the old Thread.pm

**SEE ALSO**

*threads*, *perlthrtut*, <http://www.perl.com/pub/a/2002/06/11/threads.html>

**NAME**

threads – Perl extension allowing use of interpreter based threads from perl

**SYNOPSIS**

```
use threads;

sub start_thread {
 print "Thread started\n";
}

my $thread = threads->create("start_thread", "argument");
my $thread2 = $thread->create(sub { print "I am a thread" }, "argument");
my $thread3 = async { foreach (@files) { ... } };

$thread->join();
$thread->detach();

$thread = threads->self();
$thread = threads->object($tid);

$thread->tid();
threads->tid();
threads->self->tid();

threads->yield();

threads->list();
```

**DESCRIPTION**

Perl 5.6 introduced something called interpreter threads. Interpreter threads are different from "5005threads" (the thread model of Perl 5.005) by creating a new perl interpreter per thread and not sharing any data or state between threads by default.

Prior to perl 5.8 this has only been available to people embedding perl and for emulating `fork()` on windows.

The threads API is loosely based on the old Thread.pm API. It is very important to note that variables are not shared between threads, all variables are per default thread local. To use shared variables one must use `threads::shared`.

It is also important to note that you must enable threads by doing `use threads` as early as possible in the script itself and that it is not possible to enable threading inside an `eval` "", `do`, `require`, or `use`. In particular, if you are intending to share variables with `threads::shared`, you must use `threads` before you use `threads::shared` and `threads` will emit a warning if you do it the other way around.

`$thread = threads->create(function, LIST)`

This will create a new thread with the entry point function and give it LIST as parameters. It will return the corresponding threads object. The `new()` method is an alias for `create()`.

`$thread->join`

This will wait for the corresponding thread to join. When the thread finishes, `join()` will return the return values of the entry point function. If the thread has been detached, an error will be thrown. If the program exits without all other threads having been either joined or detached, then a warning will be issued. (A program exits either because one of its threads explicitly calls `exit()`, or in the case of the main thread, reaches the end of the main program file.)

`$thread->detach`

Will make the thread unjoinable, and cause any eventual return value to be discarded.

**threads-self**

This will return the thread object for the current thread.

**\$thread-tid**

This will return the id of the thread. Thread IDs are integers, with the main thread in a program being 0. Currently Perl assigns a unique tid to every thread ever created in your program, assigning the first thread to be created a tid of 1, and increasing the tid by 1 for each new thread that's created.

NB the class method `< threads-tid()` is a quick way to get the current thread id if you don't have your thread object handy.

**threads-object( tid )**

This will return the thread object for the thread associated with the specified tid. Returns undef if there is no thread associated with the tid or no tid is specified or the specified tid is undef.

**threads-yield();**

This is a suggestion to the OS to let this thread yield CPU time to other threads. What actually happens is highly dependent upon the underlying thread implementation.

You may do `use threads qw(yield)` then use just a bare `yield` in your code.

**threads-list();**

This will return a list of all non joined, non detached threads.

**async BLOCK;**

`async` creates a thread to execute the block immediately following it. This block is treated as an anonymous sub, and so must have a semi-colon after the closing brace. Like `< threads-new`, `async` returns a thread object.

**WARNINGS****A thread exited while %d other threads were still running**

A thread (not necessarily the main thread) exited while there were still other threads running. Usually it's a good idea to first collect the return values of the created threads by joining them, and only then exit from the main thread.

**TODO**

The current implementation of threads has been an attempt to get a correct threading system working that could be built on, and optimized, in newer versions of perl.

Currently the overhead of creating a thread is rather large, also the cost of returning values can be large. These are areas where there most likely will be work done to optimize what data that needs to be cloned.

**BUGS****Parent-Child threads.**

On some platforms it might not be possible to destroy "parent" threads while there are still existing child "threads".

This will possibly be fixed in later versions of perl.

**tid is l32**

The thread id is a 32 bit integer, it can potentially overflow. This might be fixed in a later version of perl.

**Returning objects**

When you return an object the entire stash that the object is blessed as well. This will lead to a large memory usage. The ideal situation would be to detect the original stash if it existed.

### Creating threads inside BEGIN blocks

Creating threads inside BEGIN blocks (or during the compilation phase in general) does not work. (In Windows, trying to use `fork()` inside BEGIN blocks is an equally losing proposition, since it has been implemented in very much the same way as threads.)

PERL\_OLD\_SIGNALS are not threadsafe, will not be.

If your Perl has been built with PERL\_OLD\_SIGNALS (one has to explicitly add that symbol to `cflags`, see `perl -V`), signal handling is not threadsafe.

### AUTHOR and COPYRIGHT

Arthur Bergman <arthur at contiller.se>

threads is released under the same license as Perl.

Thanks to

Richard Soderberg <rs at crystalflame.net> Helping me out tons, trying to find reasons for races and other weird bugs!

Simon Cozens <simon at brecon.co.uk> Being there to answer zillions of annoying questions

Rocco Caputo <troc at netrus.net>

Vipul Ved Prakash <mail at vipul.net> Helping with debugging.

please join `perl-ithreads@perl.org` for more information

### SEE ALSO

*threads::shared*, *perlthrtut*, <http://www.perl.com/pub/a/2002/06/11/threads.html>, *perlcall*, *perlembed*, *perlguits*



**NAME**

Tie::Array – base class for tied arrays

**SYNOPSIS**

```
package Tie::NewArray;
use Tie::Array;
@ISA = ('Tie::Array');

mandatory methods
sub TIEARRAY { ... }
sub FETCH { ... }
sub FETCHSIZE { ... }

sub STORE { ... } # mandatory if elements writeable
sub STORESIZE { ... } # mandatory if elements can be added/deleted
sub EXISTS { ... } # mandatory if exists() expected to work
sub DELETE { ... } # mandatory if delete() expected to work

optional methods - for efficiency
sub CLEAR { ... }
sub PUSH { ... }
sub POP { ... }
sub SHIFT { ... }
sub UNSHIFT { ... }
sub SPLICE { ... }
sub EXTEND { ... }
sub DESTROY { ... }

package Tie::NewStdArray;
use Tie::Array;

@ISA = ('Tie::StdArray');

all methods provided by default

package main;

$object = tie @somearray, Tie::NewArray;
$object = tie @somearray, Tie::StdArray;
$object = tie @somearray, Tie::NewStdArray;
```

**DESCRIPTION**

This module provides methods for array-tying classes. See [perltie](#) for a list of the functions required in order to tie an array to a package. The basic **Tie::Array** package provides stub DESTROY, and EXTEND methods that do nothing, stub DELETE and EXISTS methods that croak() if the delete() or exists() builtins are ever called on the tied array, and implementations of PUSH, POP, SHIFT, UNSHIFT, SPLICE and CLEAR in terms of basic FETCH, STORE, FETCHSIZE, STORESIZE.

The **Tie::StdArray** package provides efficient methods required for tied arrays which are implemented as blessed references to an "inner" perl array. It inherits from **Tie::Array**, and should cause tied arrays to behave exactly like standard arrays, allowing for selective overloading of methods.

For developers wishing to write their own tied arrays, the required methods are briefly defined below. See the [perltie](#) section for more detailed descriptive, as well as example code:

**TIEARRAY classname, LIST**

The class method is invoked by the command `tie @array, classname`. Associates an array instance with the specified class. LIST would represent additional arguments (along the lines of [AnyDBM\\_File](#) and compatriots) needed to complete the association. The method should return an object of a class which provides the methods below.

**STORE** *this*, *index*, *value*

Store datum *value* into *index* for the tied array associated with object *this*. If this makes the array larger then class's mapping of `undef` should be returned for new positions.

**FETCH** *this*, *index*

Retrieve the datum in *index* for the tied array associated with object *this*.

**FETCHSIZE** *this*

Returns the total number of items in the tied array associated with object *this*. (Equivalent to `scalar(@array)`).

**STORESIZE** *this*, *count*

Sets the total number of items in the tied array associated with object *this* to be *count*. If this makes the array larger then class's mapping of `undef` should be returned for new positions. If the array becomes smaller then entries beyond *count* should be deleted.

**EXTEND** *this*, *count*

Informative call that array is likely to grow to have *count* entries. Can be used to optimize allocation. This method need do nothing.

**EXISTS** *this*, *key*

Verify that the element at index *key* exists in the tied array *this*.

The **Tie::Array** implementation is a stub that simply croaks.

**DELETE** *this*, *key*

Delete the element at index *key* from the tied array *this*.

The **Tie::Array** implementation is a stub that simply croaks.

**CLEAR** *this*

Clear (remove, delete, ...) all values from the tied array associated with object *this*.

**DESTROY** *this*

Normal object destructor method.

**PUSH** *this*, *LIST*

Append elements of *LIST* to the array.

**POP** *this*

Remove last element of the array and return it.

**SHIFT** *this*

Remove the first element of the array (shifting other elements down) and return it.

**UNSHIFT** *this*, *LIST*

Insert *LIST* elements at the beginning of the array, moving existing elements up to make room.

**SPLICE** *this*, *offset*, *length*, *LIST*

Perform the equivalent of `splice` on the array.

*offset* is optional and defaults to zero, negative values count back from the end of the array.

*length* is optional and defaults to rest of the array.

*LIST* may be empty.

Returns a list of the original *length* elements at *offset*.

**CAVEATS**

There is no support at present for tied @ISA. There is a potential conflict between magic entries needed to notice setting of @ISA, and those needed to implement 'tie'.

Very little consideration has been given to the behaviour of tied arrays when \$[ is not default value of zero.

**AUTHOR**

Nick Ing-Simmons <nik@tiuk.ti.com>

**NAME**

Tie::File – Access the lines of a disk file via a Perl array

**SYNOPSIS**

```
This file documents Tie::File version 0.93

tie @array, 'Tie::File', filename or die ...;

$array[13] = 'blah'; # line 13 of the file is now 'blah'
print $array[42]; # display line 42 of the file

$n_recs = @array; # how many records are in the file?
$#array -= 2; # chop two records off the end

for (@array) {
 s/PERL/Perl/g; # Replace PERL with Perl everywhere in the file
}

These are just like regular push, pop, unshift, shift, and splice
Except that they modify the file in the way you would expect

push @array, new_recs...;
my $r1 = pop @array;
unshift @array, new_recs...;
my $r1 = shift @array;
@old_recs = splice @array, 3, 7, new_recs...;

untie @array; # all finished
```

**DESCRIPTION**

Tie::File represents a regular text file as a Perl array. Each element in the array corresponds to a record in the file. The first line of the file is element 0 of the array; the second line is element 1, and so on.

The file is *not* loaded into memory, so this will work even for gigantic files.

Changes to the array are reflected in the file immediately.

Lazy people and beginners may now stop reading the manual.

**recsep**

What is a 'record'? By default, the meaning is the same as for the `<...>` operator: It's a string terminated by `$/`, which is probably `"\n"`. (Minor exception: on dos and Win32 systems, a 'record' is a string terminated by `"\r\n"`.) You may change the definition of "record" by supplying the `recsep` option in the `tie` call:

```
tie @array, 'Tie::File', $file, recsep => 'es';
```

This says that records are delimited by the string `es`. If the file contained the following data:

```
Curse these pesky flies!\n
```

then the `@array` would appear to have four elements:

```
"Curse th"
"e p"
"ky fli"
"!\n"
```

An undefined value is not permitted as a record separator. Perl's special "paragraph mode" semantics (à la `$/ = ""`) are not emulated.

Records read from the tied array do not have the record separator string on the end; this is to allow

```
$array[17] .= "extra";
```

to work as expected.

(See *"autochomp"*, below.) Records stored into the array will have the record separator string appended before they are written to the file, if they don't have one already. For example, if the record separator string is `"\n"`, then the following two lines do exactly the same thing:

```
$array[17] = "Cherry pie";
$array[17] = "Cherry pie\n";
```

The result is that the contents of line 17 of the file will be replaced with "Cherry pie"; a newline character will separate line 17 from line 18. This means that this code will do nothing:

```
chomp $array[17];
```

Because the chomped value will have the separator reattached when it is written back to the file. There is no way to create a file whose trailing record separator string is missing.

Inserting records that *contain* the record separator string is not supported by this module. It will probably produce a reasonable result, but what this result will be may change in a future version. Use `'splice'` to insert records or to replace one record with several.

### autochomp

Normally, array elements have the record separator removed, so that if the file contains the text

```
Gold
Frankincense
Myrrh
```

the tied array will appear to contain `("Gold", "Frankincense", "Myrrh")`. If you set `autochomp` to a false value, the record separator will not be removed. If the file above was tied with

```
tie @gifts, "Tie::File", $gifts, autochomp => 0;
```

then the array `@gifts` would appear to contain `("Gold\n", "Frankincense\n", "Myrrh\n")`, or (on Win32 systems) `("Gold\r\n", "Frankincense\r\n", "Myrrh\r\n")`.

### mode

Normally, the specified file will be opened for read and write access, and will be created if it does not exist. (That is, the flags `O_RDWR` | `O_CREAT` are supplied in the open call.) If you want to change this, you may supply alternative flags in the mode option. See *Fcntl* for a listing of available flags. For example:

```
open the file if it exists, but fail if it does not exist
use Fcntl 'O_RDWR';
tie @array, 'Tie::File', $file, mode => O_RDWR;

create the file if it does not exist
use Fcntl 'O_RDWR', 'O_CREAT';
tie @array, 'Tie::File', $file, mode => O_RDWR | O_CREAT;

open an existing file in read-only mode
use Fcntl 'O_RDONLY';
tie @array, 'Tie::File', $file, mode => O_RDONLY;
```

Opening the data file in write-only or append mode is not supported.

### memory

This is an upper limit on the amount of memory that `Tie::File` will consume at any time while managing the file. This is used for two things: managing the *read cache* and managing the *deferred write buffer*.

Records read in from the file are cached, to avoid having to re-read them repeatedly. If you read the same record twice, the first time it will be stored in memory, and the second time it will be fetched from the *read*

*cache*. The amount of data in the read cache will not exceed the value you specified for memory. If `Tie::File` wants to cache a new record, but the read cache is full, it will make room by expiring the least-recently visited records from the read cache.

The default memory limit is 2Mib. You can adjust the maximum read cache size by supplying the `memory` option. The argument is the desired cache size, in bytes.

```
I have a lot of memory, so use a large cache to speed up access
tie @array, 'Tie::File', $file, memory => 20_000_000;
```

Setting the memory limit to 0 will inhibit caching; records will be fetched from disk every time you examine them.

The memory value is not an absolute or exact limit on the memory used. `Tie::File` objects contains some structures besides the read cache and the deferred write buffer, whose sizes are not charged against memory.

### dw\_size

(This is an advanced feature. Skip this section on first reading.)

If you use deferred writing (See ["Deferred Writing"](#), below) then data you write into the array will not be written directly to the file; instead, it will be saved in the *deferred write buffer* to be written out later. Data in the deferred write buffer is also charged against the memory limit you set with the `memory` option.

You may set the `dw_size` option to limit the amount of data that can be saved in the deferred write buffer. This limit may not exceed the total memory limit. For example, if you set `dw_size` to 1000 and `memory` to 2500, that means that no more than 1000 bytes of deferred writes will be saved up. The space available for the read cache will vary, but it will always be at least 1500 bytes (if the deferred write buffer is full) and it could grow as large as 2500 bytes (if the deferred write buffer is empty.)

If you don't specify a `dw_size`, it defaults to the entire memory limit.

### Option Format

`-mode` is a synonym for `mode`. `-recsep` is a synonym for `recsep`. `-memory` is a synonym for `memory`. You get the idea.

### Public Methods

The `tie` call returns an object, say `$o`. You may call

```
$rec = $o->FETCH($n);
$o->STORE($n, $rec);
```

to fetch or store the record at line `$n`, respectively; similarly the other tied array methods. (See [perl tie](#) for details.) You may also call the following methods on this object:

### flock

```
$o->flock(MODE)
```

will lock the tied file. `MODE` has the same meaning as the second argument to the Perl built-in `flock` function; for example `LOCK_SH` or `LOCK_EX` | `LOCK_NB`. (These constants are provided by the use `Fcntl` `':flock'` declaration.)

`MODE` is optional; the default is `LOCK_EX`.

`Tie::File` promises that the following sequence of operations will be safe:

```
my $o = tie @array, "Tie::File", $filename;
$o->flock;
```

In particular, `Tie::File` will *not* read or write the file during the `tie` call. (Exception: Using `mode => O_TRUNC` will, of course, erase the file during the `tie` call. If you want to do this safely, then open the file without `O_TRUNC`, lock the file, and use `@array = ()`.)

The best way to unlock a file is to discard the object and untie the array. It is probably unsafe to unlock the file without also untying it, because if you do, changes may remain unwritten inside the object. That is why there is no shortcut for unlocking. If you really want to unlock the file prematurely, you know what to do; if you don't know what to do, then don't do it.

All the usual warnings about file locking apply here. In particular, note that file locking in Perl is **advisory**, which means that holding a lock will not prevent anyone else from reading, writing, or erasing the file; it only prevents them from getting another lock at the same time. Locks are analogous to green traffic lights: If you have a green light, that does not prevent the idiot coming the other way from plowing into you sideways; it merely guarantees to you that the idiot does not also have a green light at the same time.

#### **autochomp**

```
my $old_value = $o->autochomp(0); # disable autochomp option
my $old_value = $o->autochomp(1); # enable autochomp option

my $ac = $o->autochomp(); # recover current value
```

See *"autochomp"*, above.

#### **defer, flush, discard, and autodefer**

See *"Deferred Writing"*, below.

#### **Tying to an already-opened filehandle**

If `$fh` is a filehandle, such as is returned by `IO::File` or one of the other IO modules, you may use:

```
tie @array, 'Tie::File', $fh, ...;
```

Similarly if you opened that handle `FH` with regular `open` or `sysopen`, you may use:

```
tie @array, 'Tie::File', *FH, ...;
```

Handles that were opened write-only won't work. Handles that were opened read-only will work as long as you don't try to modify the array. Handles must be attached to seekable sources of data—that means no pipes or sockets. If `Tie::File` can detect that you supplied a non-seekable handle, the `tie` call will throw an exception. (On Unix systems, it can detect this.)

#### **Deferred Writing**

(This is an advanced feature. Skip this section on first reading.)

Normally, modifying a `Tie::File` array writes to the underlying file immediately. Every assignment like `$a[3] = ...` rewrites as much of the file as is necessary; typically, everything from line 3 through the end will need to be rewritten. This is the simplest and most transparent behavior. Performance even for large files is reasonably good.

However, under some circumstances, this behavior may be excessively slow. For example, suppose you have a million-record file, and you want to do:

```
for (@FILE) {
 $_ = "> $_";
}
```

The first time through the loop, you will rewrite the entire file, from line 0 through the end. The second time through the loop, you will rewrite the entire file from line 1 through the end. The third time through the loop, you will rewrite the entire file from line 2 to the end. And so on.

If the performance in such cases is unacceptable, you may defer the actual writing, and then have it done all at once. The following loop will perform much better for large files:

```
(tied @a)->defer;
for (@a) {
 $_ = "> $_";
}
```

```
(tied @a)->flush;
```

If `Tie::File`'s memory limit is large enough, all the writing will be done in memory. Then, when you call `->flush`, the entire file will be rewritten in a single pass.

(Actually, the preceding discussion is something of a fib. You don't need to enable deferred writing to get good performance for this common case, because `Tie::File` will do it for you automatically unless you specifically tell it not to. See ["autodefering"](#), below.)

Calling `->flush` returns the array to immediate-write mode. If you wish to discard the deferred writes, you may call `->discard` instead of `->flush`. Note that in some cases, some of the data will have been written already, and it will be too late for `->discard` to discard all the changes. Support for `->discard` may be withdrawn in a future version of `Tie::File`.

Deferred writes are cached in memory up to the limit specified by the `dw_size` option (see above). If the deferred-write buffer is full and you try to write still more deferred data, the buffer will be flushed. All buffered data will be written immediately, the buffer will be emptied, and the now-empty space will be used for future deferred writes.

If the deferred-write buffer isn't yet full, but the total size of the buffer and the read cache would exceed the memory limit, the oldest records will be expired from the read cache until the total size is under the limit.

`push`, `pop`, `shift`, `unshift`, and `splice` cannot be deferred. When you perform one of these operations, any deferred data is written to the file and the operation is performed immediately. This may change in a future version.

If you resize the array with deferred writing enabled, the file will be resized immediately, but deferred records will not be written. This has a surprising consequence: `@a = ( ... )` erases the file immediately, but the writing of the actual data is deferred. This might be a bug. If it is a bug, it will be fixed in a future version.

## Autodefering

`Tie::File` tries to guess when deferred writing might be helpful, and to turn it on and off automatically.

```
for (@a) {
 $_ = "> $_";
}
```

In this example, only the first two assignments will be done immediately; after this, all the changes to the file will be deferred up to the user-specified memory limit.

You should usually be able to ignore this and just use the module without thinking about deferring. However, special applications may require fine control over which writes are deferred, or may require that all writes be immediate. To disable the autodeferment feature, use

```
(tied @o)->autodefer(0);
```

or

```
tie @array, 'Tie::File', $file, autodefer => 0;
```

Similarly, `->autodefer(1)` re-enables autodeferment, and `->autodefer()` recovers the current value of the autodefer setting.

## CAVEATS

(That's Latin for 'warnings'.)

- This is BETA RELEASE SOFTWARE. It may have bugs. See the discussion below about the (lack of any) warranty.

In particular, this means that the interface may change in incompatible ways from one version to the next, without warning. That has happened at least once already. The interface will freeze before Perl 5.8 is released, probably sometime in April 2002.



- Reasonable effort was made to make this module efficient. Nevertheless, changing the size of a record in the middle of a large file will always be fairly slow, because everything after the new record must be moved.
- The behavior of tied arrays is not precisely the same as for regular arrays. For example:

```
This DOES print "How unusual!"
undef $a[10]; print "How unusual!\n" if defined $a[10];
```

undef-ing a Tie::File array element just blanks out the corresponding record in the file. When you read it back again, you'll get the empty string, so the supposedly-undef'ed value will be defined. Similarly, if you have `autochomp` disabled, then

```
This DOES print "How unusual!" if 'autochomp' is disabled
undef $a[10];
print "How unusual!\n" if $a[10];
```

Because when `autochomp` is disabled, `$a[10]` will read back as `"\n"` (or whatever the record separator string is.)

There are other minor differences, particularly regarding `exists` and `delete`, but in general, the correspondence is extremely close.

- Not quite every effort was made to make this module as efficient as possible. `FETCHSIZE` should use binary search instead of linear search.

The performance of the `flush` method could be improved. At present, it still rewrites the tail of the file once for each block of contiguous lines to be changed. In the typical case, this will result in only one rewrite, but in peculiar cases it might be bad. It should be possible to perform *all* deferred writing with a single rewrite.

Profiling suggests that these defects are probably minor; in any event, they will be fixed in a future version of the module.

- I have supposed that since this module is concerned with file I/O, almost all normal use of it will be heavily I/O bound. This means that the time to maintain complicated data structures inside the module will be dominated by the time to actually perform the I/O. When there was an opportunity to spend CPU time to avoid doing I/O, I tried to take it.
- You might be tempted to think that deferred writing is like transactions, with `flush` as `commit` and `discard` as `rollback`, but it isn't, so don't.

## SUBCLASSING

This version promises absolutely nothing about the internals, which may change without notice. A future version of the module will have a well-defined and stable subclassing API.

## WHAT ABOUT DB\_File?

People sometimes point out that *DB\_File* will do something similar, and ask why Tie::File module is necessary.

There are a number of reasons that you might prefer Tie::File. A list is available at [http://perl.plover.com/TieFile/why-not-DB\\_File](http://perl.plover.com/TieFile/why-not-DB_File).

## AUTHOR

Mark Jason Dominus

To contact the author, send email to: [mjd-perl-tiefile@plover.com](mailto:mjd-perl-tiefile@plover.com)

To receive an announcement whenever a new version of this module is released, send a blank email message to [mjd-perl-tiefile-subscribe@plover.com](mailto:mjd-perl-tiefile-subscribe@plover.com).

The most recent version of this module, including documentation and any news of importance, will be

available at

<http://perl.plover.com/TieFile/>

## LICENSE

Tie::File version 0.93 is copyright (C) 2002 Mark Jason Dominus.

This library is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

These terms are your choice of any of (1) the Perl Artistic Licence, or (2) version 2 of the GNU General Public License as published by the Free Software Foundation, or (3) any later version of the GNU General Public License.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this library program; it should be in the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111 USA

For licensing inquiries, contact the author at:

Mark Jason Dominus  
255 S. Warnock St.  
Philadelphia, PA 19107

## WARRANTY

Tie::File version 0.93 comes with ABSOLUTELY NO WARRANTY. For details, see the license.

## THANKS

Gigantic thanks to Jarkko Hietaniemi, for agreeing to put this in the core when I hadn't written it yet, and for generally being helpful, supportive, and competent. (Usually the rule is "choose any one.") Also big thanks to Abhijit Menon-Sen for all of the same things.

Special thanks to Craig Berry and Peter Prymmer (for VMS portability help), Randy Kobes (for Win32 portability help), Clinton Pierce and Autrijus Tang (for heroic eleventh-hour Win32 testing above and beyond the call of duty), Michael G Schwern (for testing advice), and the rest of the CPAN testers (for testing generally).

Additional thanks to: Edward Avis / Gerrit Haase / Nikola Knezevic / Nick Ing-Simmons / Tassilo von Parseval / H. Dieter Pearcey / Slaven Rezić / Peter Scott / Peter Somu / Autrijus Tang (again) / Tels / Juero Wallboer

## TODO

More tests. (The cache and heap modules need more unit tests.)

Improve SPLICE algorithm to use deferred writing machinery.

Cleverer strategy for flushing deferred writes.

More tests. (Stuff I didn't think of yet.)

Paragraph mode?

Fixed-length mode. Leave-blanks mode.

Maybe an autolocking mode?

Record locking with `fcntl()`? Then the module might support an undo log and get real transactions. What a tour de force that would be.

More tests.

**NAME**

Tie::Handle, Tie::StdHandle – base class definitions for tied handles

**SYNOPSIS**

```
package NewHandle;
require Tie::Handle;

@ISA = qw(Tie::Handle);

sub READ { ... } # Provide a needed method
sub TIEHANDLE { ... } # Overrides inherited method

package main;

tie *FH, 'NewHandle';
```

**DESCRIPTION**

This module provides some skeletal methods for handle-tying classes. See [perltie](#) for a list of the functions required in tying a handle to a package. The basic **Tie::Handle** package provides a new method, as well as methods TIEHANDLE, PRINT, PRINTF and GETC.

For developers wishing to write their own tied-handle classes, the methods are summarized below. The [perltie](#) section not only documents these, but has sample code as well:

**TIEHANDLE classname, LIST**

The method invoked by the command `tie *glob, classname`. Associates a new glob instance with the specified class. *LIST* would represent additional arguments (along the lines of [AnyDBM\\_File](#) and compatriots) needed to complete the association.

**WRITE this, scalar, length, offset**

Write *length* bytes of data from *scalar* starting at *offset*.

**PRINT this, LIST**

Print the values in *LIST*

**PRINTF this, format, LIST**

Print the values in *LIST* using *format*

**READ this, scalar, length, offset**

Read *length* bytes of data into *scalar* starting at *offset*.

**READLINE this**

Read a single line

**GETC this**

Get a single character

**CLOSE this**

Close the handle

**OPEN this, filename**

(Re-)open the handle

**BINMODE this**

Specify content is binary

**EOF this**

Test for end of file.

TELL this

Return position in the file.

SEEK this, offset, whence

Position the file.

Test for end of file.

DESTROY this

Free the storage associated with the tied handle referenced by *this*. This is rarely needed, as Perl manages its memory quite well. But the option exists, should a class wish to perform specific actions upon the destruction of an instance.

## MORE INFORMATION

The [perltie](#) section contains an example of tying handles.

## COMPATIBILITY

This version of Tie::Handle is neither related to nor compatible with the Tie::Handle (3.0) module available on CPAN. It was due to an accident that two modules with the same name appeared. The namespace clash has been cleared in favor of this module that comes with the perl core in September 2000 and accordingly the version number has been bumped up to 4.0.

**NAME**

Tie::Hash, Tie::StdHash, Tie::ExtraHash – base class definitions for tied hashes

**SYNOPSIS**

```
package NewHash;
require Tie::Hash;

@ISA = (Tie::Hash);

sub DELETE { ... } # Provides needed method
sub CLEAR { ... } # Overrides inherited method

package NewStdHash;
require Tie::Hash;

@ISA = (Tie::StdHash);

All methods provided by default, define only those needing overrides
Accessors access the storage in %{$_[0]};
TIEHANDLE should return a reference to the actual storage
sub DELETE { ... }

package NewExtraHash;
require Tie::Hash;

@ISA = (Tie::ExtraHash);

All methods provided by default, define only those needing overrides
Accessors access the storage in %{$_[0][0]};
TIEHANDLE should return an array reference with the first element being
the reference to the actual storage
sub DELETE {
 $_[0][1]->('del', $_[0][0], $_[1]); # Call the report writer
 delete $_[0][0]->{$_[1]}; # $_[0]->SUPER::DELETE($_[1]) }

package main;

tie %new_hash, 'NewHash';
tie %new_std_hash, 'NewStdHash';
tie %new_extra_hash, 'NewExtraHash',
 sub {warn "Doing \U$_[1]\E of $_[2].\n"};
```

**DESCRIPTION**

This module provides some skeletal methods for hash-tying classes. See [perltie](#) for a list of the functions required in order to tie a hash to a package. The basic **Tie::Hash** package provides a new method, as well as methods TIEHASH, EXISTS and CLEAR. The **Tie::StdHash** and **Tie::ExtraHash** packages provide most methods for hashes described in [perltie](#) (the exceptions are UNTIE and DESTROY). They cause tied hashes to behave exactly like standard hashes, and allow for selective overwriting of methods. **Tie::Hash** grandfathers the new method: it is used if TIEHASH is not defined in the case a class forgets to include a TIEHASH method.

For developers wishing to write their own tied hashes, the required methods are briefly defined below. See the [perltie](#) section for more detailed descriptive, as well as example code:

**TIEHASH classname, LIST**

The method invoked by the command `tie %hash, classname`. Associates a new hash instance with the specified class. LIST would represent additional arguments (along the lines of [AnyDBM\\_File](#) and compatriots) needed to complete the association.

STORE this, key, value

Store datum *value* into *key* for the tied hash *this*.

FETCH this, key

Retrieve the datum in *key* for the tied hash *this*.

FIRSTKEY this

Return the first key in the hash.

NEXTKEY this, lastkey

Return the next key in the hash.

EXISTS this, key

Verify that *key* exists with the tied hash *this*.

The **Tie::Hash** implementation is a stub that simply croaks.

DELETE this, key

Delete the key *key* from the tied hash *this*.

CLEAR this

Clear all values from the tied hash *this*.

### Inheriting from Tie::StdHash

The accessor methods assume that the actual storage for the data in the tied hash is in the hash referenced by `tied(%tiedhash)`. Thus overwritten `TIEHANDLE` method should return a hash reference, and the remaining methods should operate on the hash referenced by the first argument:

```
package ReportHash;
our @ISA = 'Tie::StdHash';

sub TIEHASH {
 my $storage = bless {}, shift;
 warn "New ReportHash created, stored in $storage.\n";
 $storage
}
sub STORE {
 warn "Storing data with key $_[1] at $_[0].\n";
 $_[0]{$_[1]} = $_[2]
}
```

### Inheriting from Tie::ExtraHash

The accessor methods assume that the actual storage for the data in the tied hash is in the hash referenced by `(tied(%tiedhash))[0]`. Thus overwritten `TIEHANDLE` method should return an array reference with the first element being a hash reference, and the remaining methods should operate on the hash `< %{$_[0]-[0]}`:

```
package ReportHash;
our @ISA = 'Tie::StdHash';

sub TIEHASH {
 my $storage = bless {}, shift;
 warn "New ReportHash created, stored in $storage.\n";
 [$storage, @_]
}
sub STORE {
 warn "Storing data with key $_[1] at $_[0].\n";
 $_[0][0]{$_[1]} = $_[2]
}
```

The default `TIEHANDLE` method stores "extra" arguments to `tie()` starting from offset 1 in the array referenced by `tied(%tiedhash)`; this is the same storage algorithm as in `TIEHASH` subroutine above. Hence, a typical package inheriting from **Tie::ExtraHash** does not need to overwrite this method.

#### UNTIE and DESTROY

The methods `UNTIE` and `DESTROY` are not defined in **Tie::Hash**, **Tie::StdHash**, or **Tie::ExtraHash**. Tied hashes do not require presense of these methods, but if defined, the methods will be called in proper time, see [perltie](#).

If needed, these methods should be defined by the package inheriting from **Tie::Hash**, **Tie::StdHash**, or **Tie::ExtraHash**.

#### MORE INFORMATION

The packages relating to various DBM-related implementations (*DB\_File*, *NDBM\_File*, etc.) show examples of general tied hashes, as does the [Config](#) module. While these do not utilize **Tie::Hash**, they serve as good working examples.

**NAME**

Tie::Memoize – add data to hash when needed

**SYNOPSIS**

```
require Tie::Memoize;
tie %hash, 'Tie::Memoize',
 \&fetch, # The rest is optional
 $DATA, \&exists,
 {%ini_value}, {%ini_existence};
```

**DESCRIPTION**

This package allows a tied hash to autoloading its values on the first access, and to use the cached value on the following accesses.

Only read-accesses (via fetching the value or `exists`) result in calls to the functions; the modify-accesses are performed as on a normal hash.

The required arguments during `tie` are the hash, the package, and the reference to the `FETCHing` function. The optional arguments are an arbitrary scalar `$data`, the reference to the `EXISTS` function, and initial values of the hash and of the existence cache.

Both the `FETCHing` function and the `EXISTS` functions have the same signature: the arguments are `$key`, `$data`; `$data` is the same value as given as argument during `tie()`ing. Both functions should return an empty list if the value does not exist. If `EXISTS` function is different from the `FETCHing` function, it should return a `TRUE` value on success. The `FETCHing` function should return the intended value if the key is valid.

**Inheriting from Tie::Memoize**

The structure of the `tied()` data is an array reference with elements

```
0: cache of known values
1: cache of known existence of keys
2: FETCH function
3: EXISTS function
4: $data
```

The rest is for internal usage of this package. In particular, if `TIEHASH` is overwritten, it should call `SUPER::TIEHASH`.

**EXAMPLE**

```
sub slurp {
 my ($key, $dir) = shift;
 open my $h, '<', "$dir/$key" or return;
 local $/; <$h> # slurp it all
}
sub exists { my ($key, $dir) = shift; return -f "$dir/$key" }

tie %hash, 'Tie::Memoize', \&slurp, $directory, \&exists,
 { fake_file1 => $content1, fake_file2 => $content2 },
 { pretend_does_not_exists => 0, known_to_exist => 1 };
```

This example treats the slightly modified contents of `$directory` as a hash. The modifications are that the keys *fake\_file1* and *fake\_file2* fetch values `$content1` and `$content2`, and *pretend\_does\_not\_exists* will never be accessed. Additionally, the existence of *known\_to\_exist* is never checked (so if it does not exist when its content is needed, the user of `%hash` may be confused).



**BUGS**

FIRSTKEY and NEXTKEY methods go through the keys which were already read, not all the possible keys of the hash.

**AUTHOR**

Ilya Zakharevich <mailto:perl-module-hash-memoize@ilyaz.org>.

**NAME**

Tie::RefHash – use references as hash keys

**SYNOPSIS**

```
require 5.004;
use Tie::RefHash;
tie HASHVARIABLE, 'Tie::RefHash', LIST;
tie HASHVARIABLE, 'Tie::RefHash::Nestable', LIST;

untie HASHVARIABLE;
```

**DESCRIPTION**

This module provides the ability to use references as hash keys if you first tie the hash variable to this module. Normally, only the keys of the tied hash itself are preserved as references; to use references as keys in hashes-of-hashes, use Tie::RefHash::Nestable, included as part of Tie::RefHash.

It is implemented using the standard perl TIEHASH interface. Please see the tie entry in perlfunc(1) and perltie(1) for more information.

The Nestable version works by looking for hash references being stored and converting them to tied hashes so that they too can have references as keys. This will happen without warning whenever you store a reference to one of your own hashes in the tied hash.

**EXAMPLE**

```
use Tie::RefHash;
tie %h, 'Tie::RefHash';
$a = [];
$b = {};
$c = *main;
$d = \"gunk";
$e = sub { 'foo' };
%h = ($a => 1, $b => 2, $c => 3, $d => 4, $e => 5);
$a->[0] = 'foo';
$b->{foo} = 'bar';
for (keys %h) {
 print ref($_), "\n";
}

tie %h, 'Tie::RefHash::Nestable';
$h{$a}->{$b} = 1;
for (keys %h, keys %{$h{$a}}) {
 print ref($_), "\n";
}
```

**AUTHOR**

Gurusamy Sarathy      gsar@activestate.com

**VERSION**

Version 1.30

**SEE ALSO**

perl(1), perlfunc(1), perltie(1)

**NAME**

Tie::Scalar, Tie::StdScalar – base class definitions for tied scalars

**SYNOPSIS**

```
package NewScalar;
require Tie::Scalar;

@ISA = (Tie::Scalar);

sub FETCH { ... } # Provide a needed method
sub TIESCALAR { ... } # Overrides inherited method

package NewStdScalar;
require Tie::Scalar;

@ISA = (Tie::StdScalar);

All methods provided by default, so define only what needs be overridden
sub FETCH { ... }

package main;

tie $new_scalar, 'NewScalar';
tie $new_std_scalar, 'NewStdScalar';
```

**DESCRIPTION**

This module provides some skeletal methods for scalar-tying classes. See [perlty](#) for a list of the functions required in tying a scalar to a package. The basic **Tie::Scalar** package provides a new method, as well as methods TIESCALAR, FETCH and STORE. The **Tie::StdScalar** package provides all the methods specified in [perlty](#). It inherits from **Tie::Scalar** and causes scalars tied to it to behave exactly like the built-in scalars, allowing for selective overloading of methods. The new method is provided as a means of grandfathering, for classes that forget to provide their own TIESCALAR method.

For developers wishing to write their own tied-scalar classes, the methods are summarized below. The [perlty](#) section not only documents these, but has sample code as well:

**TIESCALAR** classname, LIST

The method invoked by the command `tie $scalar, classname`. Associates a new scalar instance with the specified class. LIST would represent additional arguments (along the lines of [AnyDBM\\_File](#) and `compatriots`) needed to complete the association.

**FETCH** this

Retrieve the value of the tied scalar referenced by *this*.

**STORE** this, value

Store data *value* in the tied scalar referenced by *this*.

**DESTROY** this

Free the storage associated with the tied scalar referenced by *this*. This is rarely needed, as Perl manages its memory quite well. But the option exists, should a class wish to perform specific actions upon the destruction of an instance.

**MORE INFORMATION**

The [perlty](#) section uses a good example of tying scalars by associating process IDs with priority.

**NAME**

Tie::SubstrHash – Fixed-table-size, fixed-key-length hashing

**SYNOPSIS**

```
require Tie::SubstrHash;

tie %myhash, 'Tie::SubstrHash', $key_len, $value_len, $table_size;
```

**DESCRIPTION**

The **Tie::SubstrHash** package provides a hash-table-like interface to an array of determinate size, with constant key size and record size.

Upon tying a new hash to this package, the developer must specify the size of the keys that will be used, the size of the value fields that the keys will index, and the size of the overall table (in terms of key-value pairs, not size in hard memory). *These values will not change for the duration of the tied hash.* The newly-allocated hash table may now have data stored and retrieved. Efforts to store more than `$table_size` elements will result in a fatal error, as will efforts to store a value not exactly `$value_len` characters in length, or reference through a key not exactly `$key_len` characters in length. While these constraints may seem excessive, the result is a hash table using much less internal memory than an equivalent freely-allocated hash table.

**CAVEATS**

Because the current implementation uses the table and key sizes for the hashing algorithm, there is no means by which to dynamically change the value of any of the initialization parameters.

The hash does not support `exists()`.

## NAME

Time::gmtime – by-name interface to Perl's built-in `gmtime()` function

## SYNOPSIS

```
use Time::gmtime;
$gm = gmtime();
printf "The day in Greenwich is %s\n",
 (qw(Sun Mon Tue Wed Thu Fri Sat Sun))[gm->wday()];

use Time::gmtime w(:FIELDS);
printf "The day in Greenwich is %s\n",
 (qw(Sun Mon Tue Wed Thu Fri Sat Sun))[gm_wday()];

$now = gmctime();

use Time::gmtime;
use File::stat;
$date_string = gmctime(stat($file)->mtime);
```

## DESCRIPTION

This module's default exports override the core `gmtime()` function, replacing it with a version that returns "Time::tm" objects. This object has methods that return the similarly named structure field name from the C's tm structure from *time.h*; namely `sec`, `min`, `hour`, `mday`, `mon`, `year`, `wday`, `yday`, and `isdst`.

You may also import all the structure fields directly into your namespace as regular variables using the `:FIELDS` import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `tm_` in front their method names. Thus, `$tm_obj->mday()` corresponds to `$tm_mday` if you import the fields.

The `gmctime()` function provides a way of getting at the scalar sense of the original `CORE::gmtime()` function.

To access this functionality without the core overrides, pass the use an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

## NOTE

While this class is currently implemented using the `Class::Struct` module to build a struct-like class, you shouldn't rely upon this.

## AUTHOR

Tom Christiansen

**NAME**

Time::HiRes – High resolution alarm, sleep, gettimeofday, interval timers

**SYNOPSIS**

```
use Time::HiRes qw(usleep ualarm gettimeofday tv_interval);

usleep ($microseconds);

ualarm ($microseconds);
ualarm ($microseconds, $interval_microseconds);

$t0 = [gettimeofday];
($seconds, $microseconds) = gettimeofday;

$elapsed = tv_interval ($t0, [$seconds, $microseconds]);
$elapsed = tv_interval ($t0, [gettimeofday]);
$elapsed = tv_interval ($t0);

use Time::HiRes qw (time alarm sleep);

$now_fractions = time;
sleep ($floating_seconds);
alarm ($floating_seconds);
alarm ($floating_seconds, $floating_interval);

use Time::HiRes qw(setitimer getitimer
 ITIMER_REAL ITIMER_VIRTUAL ITIMER_PROF);

setitimer ($which, $floating_seconds, $floating_interval);
getitimer ($which);
```

**DESCRIPTION**

The Time::HiRes module implements a Perl interface to the usleep, ualarm, gettimeofday, and setitimer/getitimer system calls. See the EXAMPLES section below and the test scripts for usage; see your system documentation for the description of the underlying usleep, ualarm, gettimeofday, and setitimer/getitimer calls.

If your system lacks gettimeofday(2) or an emulation of it you don't get gettimeofday() or the one-arg form of tv\_interval(). If you don't have usleep(3) or select(2) you don't get usleep() or sleep().

If your system don't have ualarm(3) or setitimer(2) you don't get ualarm() or alarm(). If you try to import an unimplemented function in the use statement it will fail at compile time.

The following functions can be imported from this module. No functions are exported by default.

**gettimeofday ( )**

In array context returns a 2 element array with the seconds and microseconds since the epoch. In scalar context returns floating seconds like Time::HiRes::time() (see below).

**usleep ( \$useconds )**

Sleeps for the number of microseconds specified. Returns the number of microseconds actually slept. Can sleep for more than one second unlike the usleep system call. See also Time::HiRes::sleep() below.

**ualarm ( \$useconds [, \$interval\_useconds ] )**

Issues a ualarm call; interval\_useconds is optional and will be 0 if unspecified, resulting in alarm-like behaviour.

**tv\_interval**

tv\_interval ( \$ref\_to\_gettimeofday [, \$ref\_to\_later\_gettimeofday] )

Returns the floating seconds between the two times, which should have been returned by

`gettimeofday()`. If the second argument is omitted, then the current time is used.

#### `time()`

Returns a floating seconds since the epoch. This function can be imported, resulting in a nice drop-in replacement for the `time` provided with core Perl, see the EXAMPLES below.

**NOTE 1:** this higher resolution timer can return values either less or more than the core `time()`, depending on whether your platform rounds the higher resolution timer values up, down, or to the nearest to get the core `time()`, but naturally the difference should be never more than half a second.

**NOTE 2:** Since Sunday, September 9th, 2001 at 01:46:40 AM GMT (when the `time()` seconds since epoch rolled over to 1\_000\_000\_000), the default floating point format of Perl and the seconds since epoch have conspired to produce an apparent bug: if you print the value of

`Time::HiRes::time()` you seem to be getting only five decimals, not six as promised (microseconds). Not to worry, the microseconds are there (assuming your platform supports such granularity). What is going on is that the default floating point format of Perl only outputs 15 digits. In this case that means ten digits before the decimal separator and five after. To see the microseconds you can use either `printf/sprintf` with `%.6f`, or the `gettimeofday()` function in list context, which will give you the seconds and microseconds as two separate values.

#### `sleep($floating_seconds)`

Sleeps for the specified amount of seconds. Returns the number of seconds actually slept (a floating point value). This function can be imported, resulting in a nice drop-in replacement for the `sleep` provided with perl, see the EXAMPLES below.

#### `alarm($floating_seconds[, $interval_floating_seconds])`

The SIGALRM signal is sent after the specified number of seconds. Implemented using `ualarm()`. The `$interval_floating_seconds` argument is optional and will be 0 if unspecified, resulting in `alarm()`-like behaviour. This function can be imported, resulting in a nice drop-in replacement for the `alarm` provided with perl, see the EXAMPLES below.

#### `setitimer`

`setitimer($which, $floating_seconds[, $interval_floating_seconds])`

Start up an interval timer: after a certain time, a signal arrives, and more signals may keep arriving at certain intervals. To disable a timer, use time of zero. If interval is set to zero (or unspecified), the timer is disabled **after** the next delivered signal.

Use of interval timers may interfere with `alarm()`, `sleep()`, and `usleep()`. In standard-speak the "interaction is unspecified", which means that *anything* may happen: it may work, it may not.

In scalar context, the remaining time in the timer is returned.

In list context, both the remaining time and the interval are returned.

There are three interval timers: the `$which` can be `ITIMER_REAL`, `ITIMER_VIRTUAL`, or `ITIMER_PROF`.

`ITIMER_REAL` results in `alarm()`-like behavior. Time is counted in *real time*, that is, wallclock time. SIGALRM is delivered when the timer expires.

`ITIMER_VIRTUAL` counts time in (process) *virtual time*, that is, only when the process is running. In multiprocessor/user/CPU systems this may be more or less than real or wallclock time. (This time is also known as the *user time*.) SIGVTALRM is delivered when the timer expires.

`ITIMER_PROF` counts time when either the process virtual time or when the operating system is running on behalf of the process (such as I/O). (This time is also known as the *system time*.) (Collectively these times are also known as the *CPU time*.) SIGPROF is delivered when the timer expires. SIGPROF can interrupt system calls.

The semantics of interval timers for multithreaded programs are system-specific, and some systems may support additional interval timers. See your `setitimer()` documentation.

#### `getitimer ( $which )`

Return the remaining time in the interval timer specified by `$which`.

In scalar context, the remaining time is returned.

In list context, both the remaining time and the interval are returned. The interval is always what you put in using `setitimer()`.

### EXAMPLES

```
use Time::HiRes qw(usleep ualarm gettimeofday tv_interval);

$microseconds = 750_000;
usleep $microseconds;

signal alarm in 2.5s & every .1s thereafter
ualarm 2_500_000, 100_000;

get seconds and microseconds since the epoch
($s, $usec) = gettimeofday;

measure elapsed time
(could also do by subtracting 2 gettimeofday return values)
$t0 = [gettimeofday];
do bunch of stuff here
$t1 = [gettimeofday];
do more stuff here
$t0_t1 = tv_interval $t0, $t1;

$elapsed = tv_interval ($t0, [gettimeofday]);
$elapsed = tv_interval ($t0); # equivalent code

#
replacements for time, alarm and sleep that know about
floating seconds
#
use Time::HiRes;
$now_fractions = Time::HiRes::time;
Time::HiRes::sleep (2.5);
Time::HiRes::alarm (10.6666666);

use Time::HiRes qw (time alarm sleep);
$now_fractions = time;
sleep (2.5);
alarm (10.6666666);

Arm an interval timer to go off first at 10 seconds and
after that every 2.5 seconds, in process virtual time

use Time::HiRes qw (setitimer ITIMER_VIRTUAL time);

$SIG{VTALARM} = sub { print time, "\n" };
setitimer(ITIMER_VIRTUAL, 10, 2.5);
```

### C API

In addition to the perl API described above, a C API is available for extension writers. The following C functions are available in the `modglobal` hash:

| name  | C prototype |
|-------|-------------|
| ----- | -----       |



```
Time::NVtime double (*)()
Time::U2time void (*) (UV ret[2])
```

Both functions return equivalent information (like `gettimeofday`) but with different representations. The names `NVtime` and `U2time` were selected mainly because they are operating system independent. (`gettimeofday` is Un\*x-centric.)

Here is an example of using `NVtime` from C:

```
double (*myNVtime)();
SV **svp = hv_fetch(PL_modglobal, "Time::NVtime", 12, 0);
if (!svp) croak("Time::HiRes is required");
if (!SvIOK(*svp)) croak("Time::NVtime isn't a function pointer");
myNVtime = INT2PTR(double(*)(), SvIV(*svp));
printf("The current time is: %f\n", (*myNVtime)());
```

## CAVEATS

Notice that the core `time()` maybe rounding rather than truncating. What this means that the core `time()` may be giving time one second later than `gettimeofday()`, also known as `Time::HiRes::time()`.

## AUTHORS

D. Wegscheid <wegscd@whirlpool.com> R. Schertler <roderick@argon.org> J. Hietaniemi <jhi@iki.fi> G. Aas <gisle@aas.no>

## REVISION

\$Id: HiRes.pm,v 1.20 1999/03/16 02:26:13 wegscd Exp \$

\$Log: HiRes.pm,v \$ Revision 1.20 1999/03/16 02:26:13 wegscd Add documentation for NVTime and U2Time.

Revision 1.19 1998/09/30 02:34:42 wegscd No changes, bump version.

Revision 1.18 1998/07/07 02:41:35 wegscd No changes, bump version.

Revision 1.17 1998/07/02 01:45:13 wegscd Bump version to 1.17

Revision 1.16 1997/11/13 02:06:36 wegscd version bump to accomodate HiRes.xs fix.

Revision 1.15 1997/11/11 02:17:59 wegscd POD editing, courtesy of Gisle Aas.

Revision 1.14 1997/11/06 03:14:35 wegscd Update version # for Makefile.PL and HiRes.xs changes.

Revision 1.13 1997/11/05 05:36:25 wegscd change version # for Makefile.pl and HiRes.xs changes.

Revision 1.12 1997/10/13 20:55:33 wegscd Force a new version for Makefile.PL changes.

Revision 1.11 1997/09/05 19:59:33 wegscd New version to bump version for README and Makefile.PL fixes. Fix bad RCS log.

Revision 1.10 1997/05/23 01:11:38 wegscd Conditional compilation; EXPORT\_FAIL fixes.

Revision 1.2 1996/12/30 13:28:40 wegscd Update documentation for what to do when missing `ualarm()` and friends.

Revision 1.1 1996/10/17 20:53:31 wegscd Fix `=head1` being next to `__END__` so `pod2man` works

Revision 1.0 1996/09/03 18:25:15 wegscd Initial revision

## COPYRIGHT

Copyright (c) 1996–1997 Douglas E. Wegscheid. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

**NAME**

Time::Local – efficiently compute time from local and GMT time

**SYNOPSIS**

```
$time = timelocal($sec,$min,$hour,$mday,$mon,$year);
$time = timegm($sec,$min,$hour,$mday,$mon,$year);
```

**DESCRIPTION**

These routines are the inverse of built-in perl functions `localtime()` and `gmtime()`. They accept a date as a six-element array, and return the corresponding `time(2)` value in seconds since the Epoch (Midnight, January 1, 1970). This value can be positive or negative.

It is worth drawing particular attention to the expected ranges for the values provided. The value for the day of the month is the actual day (ie 1..31), while the month is the number of months since January (0..11). This is consistent with the values returned from `localtime()` and `gmtime()`.

The `timelocal()` and `timegm()` functions perform range checking on the input `$sec`, `$min`, `$hour`, `$mday`, and `$mon` values by default. If you'd rather they didn't, you can explicitly import the `timelocal_nocheck()` and `timegm_nocheck()` functions.

```
use Time::Local 'timelocal_nocheck';

{
 # The 365th day of 1999
 print scalar localtime timelocal_nocheck 0,0,0,365,0,99;

 # The twenty thousandth day since 1970
 print scalar localtime timelocal_nocheck 0,0,0,20000,0,70;

 # And even the 10,000,000th second since 1999!
 print scalar localtime timelocal_nocheck 10000000,0,0,1,0,99;
}
```

Your mileage may vary when trying these with minutes and hours, and it doesn't work at all for months.

Strictly speaking, the year should also be specified in a form consistent with `localtime()`, i.e. the offset from 1900. In order to make the interpretation of the year easier for humans, however, who are more accustomed to seeing years as two-digit or four-digit values, the following conventions are followed:

- Years greater than 999 are interpreted as being the actual year, rather than the offset from 1900. Thus, 1963 would indicate the year Martin Luther King won the Nobel prize, not the year 2863.
- Years in the range 100..999 are interpreted as offset from 1900, so that 112 indicates 2012. This rule also applies to years less than zero (but see note below regarding date range).
- Years in the range 0..99 are interpreted as shorthand for years in the rolling "current century," defined as 50 years on either side of the current year. Thus, today, in 1999, 0 would refer to 2000, and 45 to 2045, but 55 would refer to 1955. Twenty years from now, 55 would instead refer to 2055. This is messy, but matches the way people currently think about two digit dates. Whenever possible, use an absolute four digit year instead.

The scheme above allows interpretation of a wide range of dates, particularly if 4-digit years are used.

Please note, however, that the range of dates that can be actually be handled depends on the size of an integer (`time_t`) on a given platform. Currently, this is 32 bits for most systems, yielding an approximate range from Dec 1901 to Jan 2038.

Both `timelocal()` and `timegm()` croak if given dates outside the supported range.

## IMPLEMENTATION

These routines are quite efficient and yet are always guaranteed to agree with `localtime()` and `gmtime()`. We manage this by caching the start times of any months we've seen before. If we know the start time of the month, we can always calculate any time within the month. The start times are calculated using a mathematical formula. Unlike other algorithms that do multiple calls to `gmtime()`.

`timelocal()` is implemented using the same cache. We just assume that we're translating a GMT time, and then fudge it when we're done for the timezone and daylight savings arguments. Note that the timezone is evaluated for each date because countries occasionally change their official timezones. Assuming that `localtime()` corrects for these changes, this routine will also be correct.

## BUGS

The whole scheme for interpreting two-digit years can be considered a bug.

The proclivity to `croak()` is probably a bug.

**NAME**

Time::localtime – by-name interface to Perl's built-in localtime() function

**SYNOPSIS**

```
use Time::localtime;
printf "Year is %d\n", localtime->year() + 1900;

$now = ctime();

use Time::localtime;
use File::stat;
$date_string = ctime(stat($file)->mtime);
```

**DESCRIPTION**

This module's default exports override the core localtime() function, replacing it with a version that returns "Time::tm" objects. This object has methods that return the similarly named structure field name from the C's tm structure from *time.h*; namely sec, min, hour, mday, mon, year, wday, yday, and isdst.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding tm\_ in front their method names. Thus, \$tm\_obj->mday() corresponds to \$tm\_mday if you import the fields.

The ctime() function provides a way of getting at the scalar sense of the original CORE::localtime() function.

To access this functionality without the core overrides, pass the use an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the CORE::pseudo-package.

**NOTE**

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen

**NAME**

Time::tm – internal object used by Time::gmtime and Time::localtime

**SYNOPSIS**

Don't use this module directly.

**DESCRIPTION**

This module is used internally as a base class by Time::localtime And Time::gmtime functions. It creates a Time::tm struct object which is addressable just like's C's tm structure from *time.h*; namely with sec, min, hour, mday, mon, year, wday, yday, and isdst.

This class is an internal interface only.

**AUTHOR**

Tom Christiansen

**NAME**

Unicode::Collate – Unicode Collation Algorithm

**SYNOPSIS**

```
use Unicode::Collate;

#construct
$Collator = Unicode::Collate->new(%tailoring);

#sort
@sorted = $Collator->sort(@not_sorted);

#compare
$result = $Collator->cmp($a, $b); # returns 1, 0, or -1.
```

**DESCRIPTION****Constructor and Tailoring**

The new method returns a collator object.

```
$Collator = Unicode::Collate->new(
 alternate => $alternate,
 backwards => $levelNumber, # or \@levelNumbers
 entry => $element,
 normalization => $normalization_form,
 ignoreName => qr/$ignoreName/,
 ignoreChar => qr/$ignoreChar/,
 katakana_before_hiragana => $bool,
 level => $collationLevel,
 overrideCJK => \&overrideCJK,
 overrideHangul => \&overrideHangul,
 preprocess => \&preprocess,
 rearrange => \@charList,
 table => $filename,
 undefName => qr/$undefName/,
 undefChar => qr/$undefChar/,
 upper_before_lower => $bool,
);
if %tailoring is false (i.e. empty),
$Collator should do the default collation.
```

**alternate**

— see 3.2.2 Alternate Weighting, UTR #10.

This key allows to alternate weighting for variable collation elements, which are marked with an **ASTERISK** in the table (NOTE: Many punctuation marks and symbols are variable in *allkeys.txt*).

```
alternate => 'blanked', 'non-ignorable', 'shifted', or 'shift-trimmed'.
```

These names are case-insensitive. By default (if specification is omitted), 'shifted' is adopted.

|                 |                                                                                                                                                                       |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 'Blanked'       | Variable elements are ignorable at levels 1 through 3; considered at the 4th level.                                                                                   |
| 'Non-ignorable' | Variable elements are not reset to ignorable.                                                                                                                         |
| 'Shifted'       | Variable elements are ignorable at levels 1 through 3 their level 4 weight is replaced by the old level 1 weight. Level 4 weight for Non-Variable elements is 0xFFFF. |

'Shift-Trimmed' Same as 'shifted', but all FFFF's at the 4th level are trimmed.

### backwards

— see 3.1.2 French Accents, UTR #10.

```
backwards => $levelNumber or \@levelNumbers
```

Weights in reverse order; ex. level 2 (diacritic ordering) in French. If omitted, forwards at all the levels.

### entry

— see 3.1 Linguistic Features; 3.2.1 File Format, UTR #10.

Overrides a default order or defines additional collation elements

```
entry => <<'ENTRIES', # use the UCA file format
00E6 ; [0861.0020.0002.00E6] [08B1.0020.0002.00E6] # ligature <ae as <a<e 0063 0068 ;
[0893.0020.0002.0063] # "ch" in traditional Spanish 0043 0068 ; [0893.0020.0008.0043] #
"Ch" in traditional Spanish ENTRIES
```

### ignoreName

#### ignoreChar

— see Completely Ignorable, 3.2.2 Alternate Weighting, UTR #10.

Makes the entry in the table ignorable. If a collation element is ignorable, it is ignored as if the element had been deleted from there.

E.g. when 'a' and 'e' are ignorable, 'element' is equal to 'lament' (or 'lmnt').

### level

— see 4.3 Form a sort key for each string, UTR #10.

Set the maximum level. Any higher levels than the specified one are ignored.

```
Level 1: alphabetic ordering
Level 2: diacritic ordering
Level 3: case ordering
Level 4: tie-breaking (e.g. in the case when alternate is 'shifted')

ex.level => 2,
```

If omitted, the maximum is the 4th.

### normalization

— see 4.1 Normalize each input string, UTR #10.

If specified, strings are normalized before preparation of sort keys (the normalization is executed after preprocess).

As a form name, one of the following names must be used.

```
'C' or 'NFC' for Normalization Form C
'D' or 'NFD' for Normalization Form D
'KC' or 'NFKC' for Normalization Form KC
'KD' or 'NFKD' for Normalization Form KD
```

If omitted, the string is put into Normalization Form D.

If undef is passed explicitly as the value for this key, any normalization is not carried out (this may make tailoring easier if any normalization is not desired).

see **CAVEAT**.

**overrideCJK**

— see 7.1 Derived Collation Elements, UTR #10.

By default, mapping of CJK Unified Ideographs uses the Unicode codepoint order. But the mapping of CJK Unified Ideographs may be overridden.

ex. CJK Unified Ideographs in the JIS code point order.

```
overrideCJK => sub {
 my $u = shift; # get a Unicode codepoint
 my $b = pack('n', $u); # to UTF-16BE
 my $s = your_unicode_to_sjis_converter($b); # convert
 my $n = unpack('n', $s); # convert sjis to short
 [$n, 0x20, 0x2, $u]; # return the collation element
},
```

ex. ignores all CJK Unified Ideographs.

```
overrideCJK => sub { () }, # CODEREF returning empty list

where ->eq("Pe\x{4E00}rl", "Perl") is true
as U+4E00 is a CJK Unified Ideograph and to be ignorable.
```

If undef is passed explicitly as the value for this key, weights for CJK Unified Ideographs are treated as undefined. But assignment of weight for CJK Unified Ideographs in table or [entry](#) is still valid.

**overrideHangul**

— see 7.1 Derived Collation Elements, UTR #10.

By default, Hangul Syllables are decomposed into Hangul Jamo. But the mapping of Hangul Syllables may be overridden.

This tag works like [overrideCJK](#), so see there for examples.

If you want to override the mapping of Hangul Syllables, the Normalization Forms D and KD are not appropriate (they will be decomposed before overriding).

If undef is passed explicitly as the value for this key, weight for Hangul Syllables is treated as undefined without decomposition into Hangul Jamo. But definition of weight for Hangul Syllables in table or [entry](#) is still valid.

**preprocess**

— see 5.1 Preprocessing, UTR #10.

If specified, the coderef is used to preprocess before the formation of sort keys.

ex. dropping English articles, such as "a" or "the". Then, "the pen" is before "a pencil".

```
preprocess => sub {
 my $str = shift;
 $str =~ s/\b(?:an?|the)\s+//gi;
 $str;
},
```

**rearrange**

— see 3.1.3 Rearrangement, UTR #10.

Characters that are not coded in logical order and to be rearranged. By default,

```
rearrange => [0x0E40..0x0E44, 0x0EC0..0x0EC4],
```

If you want to disallow any rearrangement, pass undef or [] (a reference to an empty list) as the value for this key.



**table**

— see 3.2 Default Unicode Collation Element Table, UTR #10.

You can use another element table if desired. The table file must be in your `lib/Unicode/Collate` directory.

By default, the file `lib/Unicode/Collate/allkeys.txt` is used.

If `undef` is passed explicitly as the value for this key, no file is read (but you can define collation elements via [entry](#)).

A typical way to define a collation element table without any file of table:

```
$onlyABC = Unicode::Collate->new(
 table => undef,
 entry => << 'ENTRIES',
 0061 ; [.0101.0020.0002.0061] # LATIN SMALL LETTER A 0041 ; [.0101.0020.0008.0041] #
 LATIN CAPITAL LETTER A 0062 ; [.0102.0020.0002.0062] # LATIN SMALL LETTER B 0042 ;
 [.0102.0020.0008.0042] # LATIN CAPITAL LETTER B 0063 ; [.0103.0020.0002.0063] # LATIN
 SMALL LETTER C 0043 ; [.0103.0020.0008.0043] # LATIN CAPITAL LETTER C ENTRIES
);
```

**undefName****undefChar**

— see 6.3.4 Reducing the Repertoire, UTR #10.

Undefines the collation element as if it were unassigned in the table. This reduces the size of the table. If an unassigned character appears in the string to be collated, the sort key is made from its codepoint as a single-character collation element, as it is greater than any other assigned collation elements (in the codepoint order among the unassigned characters). But, it'd be better to ignore characters unfamiliar to you and maybe never used.

**katakana\_before\_hiragana****upper\_before\_lower**

— see 6.6 Case Comparisons; 7.3.1 Tertiary Weight Table, UTR #10.

By default, lowercase is before uppercase and hiragana is before katakana.

If the tag is made true, this is reversed.

**NOTE:** These tags simply assume any lowercase/uppercase or hiragana/katakana distinctions should occur in level 3, and their weights at level 3 should be same as those mentioned in 7.3.1, UTR #10. If you define your collation elements which violates this, these tags doesn't work validly.

**Methods for Collation**

```
@sorted = $Collator->sort(@not_sorted)
```

Sorts a list of strings.

```
$result = $Collator->cmp($a, $b)
```

Returns 1 (when `$a` is greater than `$b`) or 0 (when `$a` is equal to `$b`) or -1 (when `$a` is lesser than `$b`).

```
$result = $Collator->eq($a, $b)
```

```
$result = $Collator->ne($a, $b)
```

```
$result = $Collator->lt($a, $b)
```

```
$result = $Collator->le($a, $b)
```

```
$result = $Collator->gt($a, $b)
```

```
$result = $Collator->ge($a, $b)
```

They works like the same name operators as theirs.

```

eq : whether $a is equal to $b.
ne : whether $a is not equal to $b.
lt : whether $a is lesser than $b.
le : whether $a is lesser than $b or equal to $b.
gt : whether $a is greater than $b.
ge : whether $a is greater than $b or equal to $b.

```

```
$sortKey = $Collator->getSortKey($string)
```

— see 4.3 Form a sort key for each string, UTR #10.

Returns a sort key.

You compare the sort keys using a binary comparison and get the result of the comparison of the strings using UCA.

```

$Collator->getSortKey($a) cmp $Collator->getSortKey($b)

is equivalent to

$Collator->cmp($a, $b)

```

```
$sortKeyForm = $Collator->viewSortKey($string)
```

Returns a string formalized to display a sort key. Weights are enclosed with '[' and ']' and level boundaries are denoted by '| '.

```

use Unicode::Collate;
my $c = Unicode::Collate->new();
print $c->viewSortKey("Perl"), "\n";

output:
[09B3 08B1 09CB 094F|0020 0020 0020 0020|0008 0002 0002 0002|FFFF FFFF FFFF
Level 1 Level 2 Level 3 Level 4

```

```
$position = $Collator->index($string, $substring)
```

```
($position, $length) = $Collator->index($string, $substring)
```

— see 6.8 Searching, UTR #10.

If \$substring matches a part of \$string, returns the position of the first occurrence of the matching part in scalar context; in list context, returns a two-element list of the position and the length of the matching part.

**Notice** that the length of the matching part may differ from the length of \$substring.

**Note** that the position and the length are counted on the string after the process of preprocess, normalization, and rearrangement. Therefore, in case the specified string is not binary equal to the preprocessed/normalized/rearranged string, the position and the length may differ from those on the specified string. But it is guaranteed that, if matched, it returns a non-negative value as \$position.

If \$substring does not match any part of \$string, returns -1 in scalar context and an empty list in list context.

e.g. you say

```

my $Collator = Unicode::Collate->new(normalization => undef, level => 1);
my $str = "Ich mu\x{00DF} studieren.";
my $sub = "m\x{00FC}ss";
my $match;
if (my($pos,$len) = $Collator->index($str, $sub)) {
 $match = substr($str, $pos, $len);
}

```

and get "mu\x{00DF}" in `$match` since "mu" is primary equal to "mss".

## Other Methods

### UCA\_Version

Returns the version number of Unicode Technical Standard 10 this module consults.

### Base\_Unicode\_Version

Returns the version number of the Unicode Standard this module is based on.

## EXPORT

None by default.

## TODO

Unicode::Collate has not been ported to EBCDIC. The code mostly would work just fine but a decision needs to be made: how the module should work in EBCDIC? Should the low 256 characters be understood as Unicode or as EBCDIC code points? Should one be chosen or should there be a way to do either? Or should such translation be left outside the module for the user to do, for example by using `Encode::from_to()`? (or `utf8::unicode_to_native()/utf8::native_to_unicode()`?)

## CAVEAT

Use of the `normalization` parameter requires the **Unicode::Normalize** module.

If you need not it (say, in the case when you need not handle any combining characters), assign `normalization => undef` explicitly.

— see 6.5 Avoiding Normalization, UTR #10.

## BUGS

`index()` is an experimental method and its return value may be unreliable. The correct implementation for `index()` must be based on Locale-Sensitive Support: Level 3 in UTR #18, *Unicode Regular Expression Guidelines*.

See also 4.2 Locale-Dependent Graphemes in UTR #18.

## AUTHOR

SADAHIRO Tomoyuki, <SADAHIRO@cpan.org>

<http://homepage1.nifty.com/nomenclator/perl/>

Copyright (C) 2001–2002, SADAHIRO Tomoyuki. Japan. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

## SEE ALSO

<http://www.unicode.org/unicode/reports/tr10/>

Unicode Collation Algorithm – UTR #10

<http://www.unicode.org/unicode/reports/tr10/allkeys.txt>

The Default Unicode Collation Element Table

<http://www.unicode.org/unicode/reports/tr15/>

Unicode Normalization Forms – UAX #15

<http://www.unicode.org/unicode/reports/tr18/>

Unicode Regular Expression Guidelines – UTR #18

*Unicode::Normalize*

**NAME**

Unicode::Normalize – Unicode Normalization Forms

**SYNOPSIS**

```
use Unicode::Normalize;

$NFD_string = NFD($string); # Normalization Form D
$NFC_string = NFC($string); # Normalization Form C
$NFKD_string = NFKD($string); # Normalization Form KD
$NFKC_string = NFKC($string); # Normalization Form KC

or

use Unicode::Normalize 'normalize';

$NFD_string = normalize('D', $string); # Normalization Form D
$NFC_string = normalize('C', $string); # Normalization Form C
$NFKD_string = normalize('KD', $string); # Normalization Form KD
$NFKC_string = normalize('KC', $string); # Normalization Form KC
```

**DESCRIPTION****Normalization Forms**

```
$NFD_string = NFD($string)
 returns the Normalization Form D (formed by canonical decomposition).

$NFC_string = NFC($string)
 returns the Normalization Form C (formed by canonical decomposition followed by canonical
 composition).

$NFKD_string = NFKD($string)
 returns the Normalization Form KD (formed by compatibility decomposition).

$NFKC_string = NFKC($string)
 returns the Normalization Form KC (formed by compatibility decomposition followed by canonical
 composition).

$normalized_string = normalize($form_name, $string)
 As $form_name, one of the following names must be given.

 'C' or 'NFC' for Normalization Form C
 'D' or 'NFD' for Normalization Form D
 'KC' or 'NFKC' for Normalization Form KC
 'KD' or 'NFKD' for Normalization Form KD
```

**Decomposition and Composition**

```
$decomposed_string = decompose($string)
$decomposed_string = decompose($string, $useCompatMapping)
 Decompose the specified string and returns the result.
```

If the second parameter (a boolean) is omitted or false, decomposes it using the Canonical Decomposition Mapping. If true, decomposes it using the Compatibility Decomposition Mapping.

The string returned is not always in NFD/NFKD. Reordering may be required.

```
$NFD_string = reorder(decompose($string)); # eq. to NFD()
$NFKD_string = reorder(decompose($string, TRUE)); # eq. to NFKD()
```

```
$reordered_string = reorder($string)
```

Reorder the combining characters and the like in the canonical ordering and returns the result.

E.g., when you have a list of NFD/NFKD strings, you can get the concatenated NFD/NFKD string from them, saying

```
$concat_NFD = reorder(join ' ', @NFD_strings);
$concat_NFKD = reorder(join ' ', @NFKD_strings);
```

```
$composed_string = compose($string)
```

Returns the string where composable pairs are composed.

E.g., when you have a NFD/NFKD string, you can get its NFC/NFKC string, saying

```
$NFC_string = compose($NFD_string);
$NFKC_string = compose($NFKD_string);
```

### Quick Check

(see Annex 8, UAX #15; *DerivedNormalizationProps.txt*)

The following functions check whether the string is in that normalization form.

The result returned will be:

```
YES The string is in that normalization form.
NO The string is not in that normalization form.
MAYBE Dubious. Maybe yes, maybe no.
```

```
$result = checkNFD($string)
returns YES (1) or NO (empty string).
```

```
$result = checkNFC($string)
returns YES (1), NO (empty string), or MAYBE (undef).
```

```
$result = checkNFKD($string)
returns YES (1) or NO (empty string).
```

```
$result = checkNFKC($string)
returns YES (1), NO (empty string), or MAYBE (undef).
```

```
$result = check($form_name, $string)
returns YES (1), NO (empty string), or MAYBE (undef).
$form_name is alike to that for normalize().
```

### Note

In the cases of NFD and NFKD, the answer must be either YES or NO. The answer MAYBE may be returned in the cases of NFC and NFKC.

A MAYBE-NFC/NFKC string should contain at least one combining character or the like. For example, COMBINING ACUTE ACCENT has the MAYBE\_NFC/MAYBE\_NFKC property. Both `checkNFC("A\N{COMBINING ACUTE ACCENT}")` and `checkNFKC("B\N{COMBINING ACUTE ACCENT}")` will return MAYBE. "A\N{COMBINING ACUTE ACCENT}" is not in NFC (its NFC is "\N{LATIN CAPITAL LETTER A WITH ACUTE}"), while "B\N{COMBINING ACUTE ACCENT}" is in NFC.

If you want to check exactly, compare the string with its NFC/NFKC; i.e.,

```
$string eq NFC($string) # more thorough than checkNFC($string)
$string eq NFKC($string) # more thorough than checkNFKC($string)
```

## Character Data

These functions are interface of character data used internally. If you want only to get Unicode normalization forms, you don't need call them yourself.

```
$canonical_decomposed = getCanon($codepoint)
```

If the character of the specified codepoint is canonically decomposable (including Hangul Syllables), returns the **completely decomposed** string canonically equivalent to it.

If it is not decomposable, returns undef.

```
$compatibility_decomposed = getCompat($codepoint)
```

If the character of the specified codepoint is compatibility decomposable (including Hangul Syllables), returns the **completely decomposed** string compatibility equivalent to it.

If it is not decomposable, returns undef.

```
$codepoint_composite = getComposite($codepoint_here, $codepoint_next)
```

If two characters here and next (as codepoints) are composable (including Hangul Jamo/Syllables and Composition Exclusions), returns the codepoint of the composite.

If they are not composable, returns undef.

```
$combining_class = getCombinClass($codepoint)
```

Returns the combining class of the character as an integer.

```
$is_exclusion = isExclusion($codepoint)
```

Returns a boolean whether the character of the specified codepoint is a composition exclusion.

```
$is_singleton = isSingleton($codepoint)
```

Returns a boolean whether the character of the specified codepoint is a singleton.

```
$is_non_startar_decomposition = isNonStDecomp($codepoint)
```

Returns a boolean whether the canonical decomposition of the character of the specified codepoint is a Non-Starter Decomposition.

```
$may_be_composed_with_prev_char = isComp2nd($codepoint)
```

Returns a boolean whether the character of the specified codepoint may be composed with the previous one in a certain composition (including Hangul Compositions, but excluding Composition Exclusions and Non-Starter Decompositions).

## EXPORT

NFC, NFD, NFKC, NFKD: by default.

normalize and other some functions: on request.

## AUTHOR

SADAHIRO Tomoyuki, <SADAHIRO@cpan.org>

<http://homepage1.nifty.com/nomenclator/perl/>

Copyright (C) 2001-2002, SADAHIRO Tomoyuki. Japan. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

## SEE ALSO

<http://www.unicode.org/unicode/reports/tr15/>

Unicode Normalization Forms – UAX #15

<http://www.unicode.org/Public/UNIDATA/DerivedNormalizationProps.txt>  
Derived Normalization Properties



**NAME**

Unicode::UCD – Unicode character database

**SYNOPSIS**

```
use Unicode::UCD 'charinfo';
my $charinfo = charinfo($codepoint);

use Unicode::UCD 'charblock';
my $charblock = charblock($codepoint);

use Unicode::UCD 'charscript';
my $charscript = charblock($codepoint);

use Unicode::UCD 'charblocks';
my $charblocks = charblocks();

use Unicode::UCD 'charscripts';
my %charscripts = charscripts();

use Unicode::UCD qw(charscript charinrange);
my $range = charscript($script);
print "looks like $script\n" if charinrange($range, $codepoint);

use Unicode::UCD 'compexcl';
my $compexcl = compexcl($codepoint);

my $unicode_version = Unicode::UCD::UnicodeVersion();
```

**DESCRIPTION**

The Unicode::UCD module offers a simple interface to the Unicode Character Database.

**charinfo**

```
use Unicode::UCD 'charinfo';
my $charinfo = charinfo(0x41);
```

`charinfo()` returns a reference to a hash that has the following fields as defined by the Unicode standard:

| key           |                                                                  |
|---------------|------------------------------------------------------------------|
| code          | code point with at least four hexdigits                          |
| name          | name of the character IN UPPER CASE                              |
| category      | general category of the character                                |
| combining     | classes used in the Canonical Ordering Algorithm                 |
| bidirectional | bidirectional category                                           |
| decomposition | character decomposition mapping                                  |
| decimal       | if decimal digit this is the integer numeric value               |
| digit         | if digit this is the numeric value                               |
| numeric       | if numeric is the integer or rational numeric value              |
| mirrored      | if mirrored in bidirectional text                                |
| unicode10     | Unicode 1.0 name if existed and different                        |
| comment       | ISO 10646 comment field                                          |
| upper         | uppercase equivalent mapping                                     |
| lower         | lowercase equivalent mapping                                     |
| title         | titlecase equivalent mapping                                     |
| block         | block the character belongs to (used in <code>\p{In...}</code> ) |
| script        | script the character belongs to                                  |

If no match is found, a reference to an empty hash is returned.

The `block` property is the same as returned by `charinfo()`. It is not defined in the Unicode Character Database proper (Chapter 4 of the Unicode 3.0 Standard, aka TUS3) but instead in an auxiliary database (Chapter 14 of TUS3). Similarly for the `script` property.

Note that you cannot do (de)composition and casing based solely on the above decomposition and lower, upper, title, properties, you will need also the `compexcl()`, `casefold()`, and `casespec()` functions.

### charblock

```
use Unicode::UCD 'charblock';

my $charblock = charblock(0x41);
my $charblock = charblock(1234);
my $charblock = charblock("0x263a");
my $charblock = charblock("U+263a");

my $range = charblock('Armenian');
```

With a **code point argument** `charblock()` returns the *block* the character belongs to, e.g. Basic Latin. Note that not all the character positions within all blocks are defined.

See also [/Blocks versus Scripts](#).

If supplied with an argument that can't be a code point, `charblock()` tries to do the opposite and interpret the argument as a character block. The return value is a *range*: an anonymous list of lists that contain *start-of-range*, *end-of-range* code point pairs. You can test whether a code point is in a range using the [/charinrange](#) function. If the argument is not a known character block, `undef` is returned.

### charscript

```
use Unicode::UCD 'charscript';

my $charscript = charscript(0x41);
my $charscript = charscript(1234);
my $charscript = charscript("U+263a");

my $range = charscript('Thai');
```

With a **code point argument** `charscript()` returns the *script* the character belongs to, e.g. Latin, Greek, Han.

See also [/Blocks versus Scripts](#).

If supplied with an argument that can't be a code point, `charscript()` tries to do the opposite and interpret the argument as a character script. The return value is a *range*: an anonymous list of lists that contain *start-of-range*, *end-of-range* code point pairs. You can test whether a code point is in a range using the [/charinrange](#) function. If the argument is not a known character script, `undef` is returned.

### charblocks

```
use Unicode::UCD 'charblocks';

my $charblocks = charblocks();
```

`charblocks()` returns a reference to a hash with the known block names as the keys, and the code point ranges (see [/charblock](#)) as the values.

See also [/Blocks versus Scripts](#).

### charscripts

```
use Unicode::UCD 'charscripts';

my %charscripts = charscripts();
```

`charscripts()` returns a hash with the known script names as the keys, and the code point ranges (see [/charscript](#)) as the values.

See also */Blocks versus Scripts*.

## Blocks versus Scripts

The difference between a block and a script is that scripts are closer to the linguistic notion of a set of characters required to present languages, while block is more of an artifact of the Unicode character numbering and separation into blocks of (mostly) 256 characters.

For example the Latin **script** is spread over several **blocks**, such as Basic Latin, Latin 1 Supplement, Latin Extended-A, and Latin Extended-B. On the other hand, the Latin script does not contain all the characters of the Basic Latin block (also known as the ASCII): it includes only the letters, and not, for example, the digits or the punctuation.

For blocks see <http://www.unicode.org/Public/UNIDATA/Blocks.txt>

For scripts see UTR #24: <http://www.unicode.org/unicode/reports/tr24/>

## Matching Scripts and Blocks

Scripts are matched with the regular-expression construct `\p{...}` (e.g. `\p{Tibetan}` matches characters of the Tibetan script), while `\p{In...}` is used for blocks (e.g. `\p{InTibetan}` matches any of the 256 code points in the Tibetan block).

## Code Point Arguments

A *code point argument* is either a decimal or a hexadecimal scalar designating a Unicode character, or U+ followed by hexadecimals designating a Unicode character. Note that Unicode is **not** limited to 16 bits (the number of Unicode characters is open-ended, in theory unlimited): you may have more than 4 hexdigits.

## charinrange

In addition to using the `\p{In...}` and `\P{In...}` constructs, you can also test whether a code point is in the *range* as returned by */charblock* and */charscript* or as the values of the hash returned by */charblocks* and */charscripts* by using `charinrange()`:

```
use Unicode::UCD qw(charscript charinrange);

$range = charscript('Hiragana');
print "looks like hiragana\n" if charinrange($range, $codepoint);
```

## compexcl

```
use Unicode::UCD 'compexcl';

my $compexcl = compexcl("09dc");
```

The `compexcl()` returns the composition exclusion (that is, if the character should not be produced during a precomposition) of the character specified by a **code point argument**.

If there is a composition exclusion for the character, true is returned. Otherwise, false is returned.

## casefold

```
use Unicode::UCD 'casefold';

my %casefold = casefold("09dc");
```

The `casefold()` returns the locale-independent case folding of the character specified by a **code point argument**.

If there is a case folding for that character, a reference to a hash with the following fields is returned:

|         |                                         |
|---------|-----------------------------------------|
| key     |                                         |
| code    | code point with at least four hexdigits |
| status  | "C", "F", "S", or "I"                   |
| mapping | one or more codes separated by spaces   |

The meaning of the *status* is as follows:

|   |                                                                                                                                                                                                                                                                                                                                                              |
|---|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| C | common case folding, common mappings shared by both simple and full mappings                                                                                                                                                                                                                                                                                 |
| F | full case folding, mappings that cause strings to grow in length. Multiple characters are separated by spaces                                                                                                                                                                                                                                                |
| S | simple case folding, mappings to single characters where different from F                                                                                                                                                                                                                                                                                    |
| I | special case for dotted uppercase I and dotless lowercase i <ul style="list-style-type: none"> <li>- If this mapping is included, the result is case-insensitive, but dotless and dotted I's are not distinguished</li> <li>- If this mapping is excluded, the result is not fully case-insensitive, but dotless and dotted I's are distinguished</li> </ul> |

If there is no case folding for that character, undef is returned.

For more information about case mappings see <http://www.unicode.org/unicode/reports/tr21/>

### casespec

```
use Unicode::UCD 'casespec';

my %casespec = casespec("09dc");
```

The `casespec()` returns the potentially locale-dependent case mapping of the character specified by a **code point argument**. The mapping may change the length of the string (which the basic Unicode case mappings as returned by `charinfo()` never do).

If there is a case folding for that character, a reference to a hash with the following fields is returned:

|           |                                         |
|-----------|-----------------------------------------|
| key       |                                         |
| code      | code point with at least four hexdigits |
| lower     | lowercase                               |
| title     | titlecase                               |
| upper     | uppercase                               |
| condition | condition list (may be undef)           |

The condition is optional. Where present, it consists of one or more *locales* or *contexts*, separated by spaces (other than as used to separate elements, spaces are to be ignored). A condition list overrides the normal behavior if all of the listed conditions are true. Case distinctions in the condition list are not significant. Conditions preceded by "NON\_" represent the negation of the condition

Note that when there are multiple case folding definitions for a single code point because of different locales, the value returned by `casespec()` is a hash reference which has the locales as the keys and hash references as described above as the values.

A *locale* is defined as a 2-letter ISO 3166 country code, possibly followed by a "\_" and a 2-letter ISO language code (possibly followed by a "\_" and a variant code). You can find the lists of those codes, see [Locale::Country](#) and [Locale::Language](#).

A *context* is one of the following choices:

|         |                                                                                           |
|---------|-------------------------------------------------------------------------------------------|
| FINAL   | The letter is not followed by a letter of general category L (e.g. Ll, Lt, Lu, Lm, or Lo) |
| MODERN  | The mapping is only used for modern text                                                  |
| AFTER_i | The last base character was "i" (U+0069)                                                  |

For more information about case mappings see <http://www.unicode.org/unicode/reports/tr21/>

**Unicode::UCD::UnicodeVersion**

`Unicode::UCD::UnicodeVersion()` returns the version of the Unicode Character Database, in other words, the version of the Unicode standard the database implements. The version is a string of numbers delimited by dots ( `' . '` ).

**Implementation Note**

The first use of `charinfo()` opens a read-only filehandle to the Unicode Character Database (the database is included in the Perl distribution). The filehandle is then kept open for further queries. In other words, if you are wondering where one of your filehandles went, that's where.

**BUGS**

Does not yet support EBCDIC platforms.

**AUTHOR**

Jarkko Hietaniemi

**NAME**

UNIVERSAL – base class for ALL classes (blessed references)

**SYNOPSIS**

```
$is_io = $fd->isa("IO::Handle");
$is_io = Class->isa("IO::Handle");

$sub = $obj->can("print");
$sub = Class->can("print");

use UNIVERSAL qw(isa can VERSION);
$yes = isa $ref, "HASH" ;
$sub = can $ref, "fandango" ;
$ver = VERSION $obj ;
```

**DESCRIPTION**

UNIVERSAL is the base class which all bless references will inherit from, see [perlobj](#).

UNIVERSAL provides the following methods and functions:

`$obj-isa( TYPE ), CLASS-isa( TYPE ), isa( VAL, TYPE )`

C<TYPE> is a package name  
 \$obj is a blessed reference or a string containing a package name  
 C<CLASS> is a package name  
 C<VAL> is any of the above or an unblessed reference

When used as an instance or class method (`$obj-isa( TYPE )`), `isa` returns *true* if `$obj` is blessed into package `TYPE` or inherits from package `TYPE`.

When used as a class method (`CLASS-isa( TYPE )`; sometimes referred to as a static method), `isa` returns *true* if `CLASS` inherits from (or is itself) the name of the package `TYPE` or inherits from package `TYPE`.

When used as a function, like

```
use UNIVERSAL qw(isa) ;
$yes = isa $h, "HASH";
$yes = isa "Foo", "Bar";
```

or

```
require UNIVERSAL ;
$yes = UNIVERSAL::isa $a, "ARRAY";
```

, `isa` returns *true* in the same cases as above and also if `VAL` is an unblessed reference to a perl variable of type `TYPE`, such as "HASH", "ARRAY", or "Regexp".

`$obj-can( METHOD ), CLASS-can( METHOD ), can( VAL, METHOD )`

`can` checks if the object or class has a method called `METHOD`. If it does then a reference to the sub is returned. If it does not then *undef* is returned. This includes methods inherited or imported by `$obj`, `CLASS`, or `VAL`.

`can` cannot know whether an object will be able to provide a method through AUTOLOAD, so a return value of *undef* does not necessarily mean the object will not be able to handle the method call. To get around this some module authors use a forward declaration (see [perlsub](#)) for methods they will handle via AUTOLOAD. For such ‘dummy’ subs, `can` will still return a code reference, which, when called, will fall through to the AUTOLOAD. If no suitable AUTOLOAD is provided, calling the coderef will cause an error.

`can` can be called as a class (static) method, an object method, or a function.

When used as a function, if VAL is a blessed reference or package name which has a method called METHOD, can returns a reference to the subroutine. If VAL is not a blessed reference, or if it does not have a method METHOD, *undef* is returned.

#### VERSION ( [ REQUIRE ] )

VERSION will return the value of the variable \$VERSION in the package the object is blessed into. If REQUIRE is given then it will do a comparison and die if the package version is not greater than or equal to REQUIRE.

VERSION can be called as either a class (static) method, an object method or or a function.

These subroutines should *not* be imported via `use UNIVERSAL qw(...)`. If you want simple local access to them you can do

```
*isa = \&UNIVERSAL::isa;
```

to import isa into your package.

**NAME**

User::grent – by-name interface to Perl's built-in `getgr*` () functions

**SYNOPSIS**

```
use User::grent;
$gr = getgrgid(0) or die "No group zero";
if ($gr->name eq 'wheel' && @{$gr->members} > 1) {
 print "gid zero name wheel, with other members";
}

use User::grent qw(:FIELDS;
getgrgid(0) or die "No group zero";
if ($gr_name eq 'wheel' && @gr_members > 1) {
 print "gid zero name wheel, with other members";
}

$gr = getgr($whoever);
```

**DESCRIPTION**

This module's default exports override the core `getgrent()`, `getgruid()`, and `getgrnam()` functions, replacing them with versions that return "User::grent" objects. This object has methods that return the similarly named structure field name from the C's `passwd` structure from *grp.h*; namely `name`, `passwd`, `gid`, and `members` (not `mem`). The first three return scalars, the last an array reference.

You may also import all the structure fields directly into your namespace as regular variables using the `:FIELDS` import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `gr_`. Thus, `$group_obj->gid()` corresponds to `$gr_gid` if you import the fields. Array references are available as regular array variables, so `@{ $group_obj->members() }` would be simply `@gr_members`.

The `getpw()` function is a simple front-end that forwards a numeric argument to `getpwuid()` and the rest to `getpwnam()`.

To access this functionality without the core overrides, pass the use an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::pseudo-package`.

**NOTE**

While this class is currently implemented using the `Class::Struct` module to build a struct-like class, you shouldn't rely upon this.

**AUTHOR**

Tom Christiansen



**NAME**

User::pwent – by-name interface to Perl's built-in `getpw*` () functions

**SYNOPSIS**

```
use User::pwent;
$pw = getpwnam('daemon') || die "No daemon user";
if ($pw->uid == 1 && $pw->dir =~ m#^(bin|tmp)?\z#s) {
 print "gid 1 on root dir";
}

$real_shell = $pw->shell || '/bin/sh';

for (($fullname, $office, $workphone, $homephone) =
 split /\s*,\s*/, $pw->gecos)
{
 s/&/ucfirst(lc($pw->name))/ge;
}

use User::pwent qw(:FIELDS);
getpwnam('daemon') || die "No daemon user";
if ($pw_uid == 1 && $pw_dir =~ m#^(bin|tmp)?\z#s) {
 print "gid 1 on root dir";
}

$pw = getpw($whoever);

use User::pwent qw/:DEFAULT pw_has/;
if (pw_has(qw[gecos expire quota])) { }
if (pw_has("name uid gid passwd")) { }
print "Your struct pwd has: ", scalar pw_has(), "\n";
```

**DESCRIPTION**

This module's default exports override the core `getpwent()`, `getpwuid()`, and `getpwnam()` functions, replacing them with versions that return `User::pwent` objects. This object has methods that return the similarly named structure field name from the C's `passwd` structure from *pwd.h*, stripped of their leading "pw\_" parts, namely `name`, `passwd`, `uid`, `gid`, `change`, `age`, `quota`, `comment`, `class`, `gecos`, `dir`, `shell`, and `expire`. The `passwd`, `gecos`, and `shell` fields are tainted when running in taint mode.

You may also import all the structure fields directly into your namespace as regular variables using the `:FIELDS` import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `pw_` in front of their method names. Thus, `< $passwd_obj-shell` corresponds to `$pw_shell` if you import the fields.

The `getpw()` function is a simple front-end that forwards a numeric argument to `getpwuid()` and the rest to `getpwnam()`.

To access this functionality without the core overrides, pass the use an empty import list, and then access function functions with their full qualified names. The built-ins are always still available via the `CORE::` pseudo-package.

**System Specifics**

Perl believes that no machine ever has more than one of `change`, `age`, or `quota` implemented, nor more than one of either `comment` or `class`. Some machines do not support `expire`, `gecos`, or allegedly, `passwd`. You may call these methods no matter what machine you're on, but they return `undef` if unimplemented.

You may ask whether one of these was implemented on the system Perl was built on by asking the importable `pw_has` function about them. This function returns true if all parameters are supported fields on

the build platform, false if one or more were not, and raises an exception if you asked about a field that Perl never knows how to provide. Parameters may be in a space-separated string, or as separate arguments. If you pass no parameters, the function returns the list of `struct pwd` fields supported by your build platform's C library, as a list in list context, or a space-separated string in scalar context. Note that just because your C library had a field doesn't necessarily mean that it's fully implemented on that system.

Interpretation of the `gecos` field varies between systems, but traditionally holds 4 comma-separated fields containing the user's full name, office location, work phone number, and home phone number. An `&` in the `gecos` field should be replaced by the user's properly capitalized login name. The `shell` field, if blank, must be assumed to be `/bin/sh`. Perl does not do this for you. The `passwd` is one-way hashed garble, not clear text, and may not be unhashed save by brute-force guessing. Secure systems use more a more secure hashing than DES. On systems supporting shadow password systems, Perl automatically returns the shadow password entry when called by a suitably empowered user, even if your underlying vendor-provided C library was too short-sighted to realize it should do this.

See `passwd(5)` and `getpwent(3)` for details.

## NOTE

While this class is currently implemented using the `Class::Struct` module to build a struct-like class, you shouldn't rely upon this.

## AUTHOR

Tom Christiansen

## HISTORY

March 18th, 2000

Reworked internals to support better interface to dodgy fields than normal Perl function provides.  
Added `pw_has()` field. Improved documentation.

## NAME

utf8 – Perl pragma to enable/disable UTF-8 (or UTF-EBCDIC) in source code

## SYNOPSIS

```
use utf8;
no utf8;
```

## DESCRIPTION

The `use utf8` pragma tells the Perl parser to allow UTF-8 in the program text in the current lexical scope (allow UTF-EBCDIC on EBCDIC based platforms). The `no utf8` pragma tells Perl to switch back to treating the source text as literal bytes in the current lexical scope.

This pragma is primarily a compatibility device. Perl versions earlier than 5.6 allowed arbitrary bytes in source code, whereas in future we would like to standardize on the UTF-8 encoding for source text. Until UTF-8 becomes the default format for source text, this pragma should be used to recognize UTF-8 in the source. When UTF-8 becomes the standard source format, this pragma will effectively become a `no-op`. For convenience in what follows the term *UTF-X* is used to refer to UTF-8 on ASCII and ISO Latin based platforms and UTF-EBCDIC on EBCDIC based platforms.

Enabling the `utf8` pragma has the following effect:

- Bytes in the source text that have their high-bit set will be treated as being part of a literal UTF-8 character. This includes most literals such as identifier names, string constants, and constant regular expression patterns.

On EBCDIC platforms characters in the Latin 1 character set are treated as being part of a literal UTF-EBCDIC character.

Note that if you have bytes with the eighth bit on in your script (for example embedded Latin-1 in your string literals), `use utf8` will be unhappy since the bytes are most probably not well-formed UTF-8. If you want to have such bytes and use `utf8`, you can disable `utf8` until the end the block (or file, if at top level) by `no utf8;`.

## Utility functions

The following functions are defined in the `utf8::` package by the perl core.

- `$num_octets = utf8::upgrade($string);`  
Converts (in-place) internal representation of string to Perl's internal *UTF-X* form. Returns the number of octets necessary to represent the string as *UTF-X*. Can be used to make sure that the UTF-8 flag is on, so that `\w` or `lc()` work as expected on strings containing characters in the range 0x80–0xFF. Note that this should not be used to convert a legacy byte encoding to Unicode: use `Encode` for that. Affected by the encoding pragma.
- `utf8::downgrade($string[, FAIL_OK])`  
Converts (in-place) internal representation of string to be un-encoded bytes. Returns true on success. On failure dies or, if the value of `FAIL_OK` is true, returns false. Can be used to make sure that the UTF-8 flag is off, e.g. when you want to make sure that the `substr()` or `length()` function works with the usually faster byte algorithm. Note that this should not be used to convert Unicode back to a legacy byte encoding: use `Encode` for that. **Not** affected by the encoding pragma.
- `utf8::encode($string)`  
Converts (in-place) *\$string* from logical characters to octet sequence representing it in Perl's *UTF-X* encoding. Same as `Encode::encode_utf8()`. Note that this should not be used to convert a legacy byte encoding to Unicode: use `Encode` for that.
- `$flag = utf8::decode($string)`  
Attempts to convert *\$string* in-place from Perl's *UTF-X* encoding into logical characters. Same as `Encode::decode_utf8()`. Note that this should not be used to convert Unicode back to a legacy

byte encoding: use Encode for that.

- `$flag = utf8::valid(String)`

[INTERNAL] Test whether `String` is in a consistent state. Will return true if string is held as bytes, or is well-formed UTF-8 and has the UTF-8 flag on. Main reason for this routine is to allow Perl's testsuite to check that operations have left strings in a consistent state.

`utf8::encode` is like `utf8::upgrade`, but the UTF8 flag is cleared. See [perlunicode](#) for more on the UTF8 flag and the C API functions `sv_utf8_upgrade`, `sv_utf8_downgrade`, `sv_utf8_encode`, and `sv_utf8_decode`, which are wrapped by the Perl functions `utf8::upgrade`, `utf8::downgrade`, `utf8::encode` and `utf8::decode`. Note that in the Perl 5.8.0 implementation the functions `utf8::valid`, `utf8::encode`, `utf8::decode`, `utf8::upgrade`, and `utf8::downgrade` are always available, without a `require utf8` statement—this may change in future releases.

## BUGS

One can have Unicode in identifier names, but not in package/class or subroutine names. While some limited functionality towards this does exist as of Perl 5.8.0, that is more accidental than designed; use of Unicode for the said purposes is unsupported.

One reason of this unfinishedness is its (currently) inherent unportability: since both package names and subroutine names may need to be mapped to file and directory names, the Unicode capability of the filesystem becomes important—and there unfortunately aren't portable answers.

## SEE ALSO

[perlunicode](#), [bytes](#)

**NAME**

vars – Perl pragma to predeclare global variable names (obsolete)

**SYNOPSIS**

```
use vars qw($frob @mung %seen);
```

**DESCRIPTION**

NOTE: For variables in the current package, the functionality provided by this pragma has been superseded by `our` declarations, available in Perl v5.6.0 or later. See [our](#).

This will predeclare all the variables whose names are in the list, allowing you to use them under "use strict", and disabling any typo warnings.

Unlike pragmas that affect the `%^H` hints variable, the `use vars` and `use subs` declarations are not BLOCK-scoped. They are thus effective for the entire file in which they appear. You may not rescind such declarations with `no vars` or `no subs`.

Packages such as the **AutoLoader** and **SelfLoader** that delay loading of subroutines within packages can create problems with package lexicals defined using `my()`. While the **vars** pragma cannot duplicate the effect of package lexicals (total transparency outside of the package), it can act as an acceptable substitute by pre-declaring global symbols, ensuring their availability to the later-loaded routines.

See [Pragmatic Modules](#).

**NAME**

vmsish – Perl pragma to control VMS-specific language features

**SYNOPSIS**

```
use vmsish;

use vmsish 'status'; # or '$?'
use vmsish 'exit';
use vmsish 'time';

use vmsish 'hushed';
no vmsish 'hushed';
vmsish::hushed($hush);

use vmsish;
no vmsish 'time';
```

**DESCRIPTION**

If no import list is supplied, all possible VMS-specific features are assumed. Currently, there are four VMS-specific features available: 'status' (a.k.a '\$?'), 'exit', 'time' and 'hushed'.

If you're not running VMS, this module does nothing.

**vmsish status**

This makes \$? and system return the native VMS exit status instead of emulating the POSIX exit status.

**vmsish exit**

This makes `exit 1` produce a successful exit (with status `SS$_NORMAL`), instead of emulating `UNIX exit()`, which considers `exit 1` to indicate an error. As with the CRTL's `exit()` function, `exit 0` is also mapped to an exit status of `SS$_NORMAL`, and any other argument to `exit()` is used directly as Perl's exit status.

**vmsish time**

This makes all times relative to the local time zone, instead of the default of Universal Time (a.k.a Greenwich Mean Time, or GMT).

**vmsish hushed**

This suppresses printing of VMS status messages to `SYS$OUTPUT` and `SYS$ERROR` if Perl terminates with an error status. and allows programs that are expecting "unix-style" Perl to avoid having to parse VMS error messages. It does not suppress any messages from Perl itself, just the messages generated by DCL after Perl exits. The DCL symbol `$STATUS` will still have the termination status, but with a high-order bit set:

**EXAMPLE:**

```
$ perl -e"exit 44;" Non-hushed error exit
%SYSTEM-F-ABORT, abort DCL message
$ show sym $STATUS
$STATUS == "%X0000002C"

$ perl -e"use vmsish qw(hushed); exit 44;" Hushed error exit
$ show sym $STATUS
$STATUS == "%X1000002C"
```

The 'hushed' flag has a global scope during compilation: the `exit()` or `die()` commands that are compiled after 'vmsish hushed' will be hushed when they are executed. Doing a "no vmsish 'hushed'" turns off the hushed flag.

The status of the hushed flag also affects output of VMS error messages from compilation errors. Again, you still get the Perl error message (and the code in `$STATUS`)

## EXAMPLE:

```
use vmsish 'hushed'; # turn on hushed flag
use Carp; # Carp compiled hushed
exit 44; # will be hushed
croak('I die'); # will be hushed
no vmsish 'hushed'; # turn off hushed flag
exit 44; # will not be hushed
croak('I die2'): # WILL be hushed, croak was compiled hushed
```

You can also control the 'hushed' flag at run-time, using the built-in routine `vmsish::hushed()`. Without argument, it returns the hushed status. Since `vmsish::hushed` is built-in, you do not need to "use vmsish" to call it.

## EXAMPLE:

```
if ($quiet_exit) {
 vmsish::hushed(1);
}
print "Ssshhhh...I'm hushed...\n" if vmsish::hushed();
exit 44;
```

Note that an `exit()` or `die()` that is compiled 'hushed' because of "use vmsish" is not un-hushed by calling `vmsish::hushed(0)` at runtime.

The messages from error exits from inside the Perl core are generally more serious, and are not suppressed.

See [Pragmatic Modules](#).

**NAME**

warnings – Perl pragma to control optional warnings

**SYNOPSIS**

```
use warnings;
no warnings;

use warnings "all";
no warnings "all";

use warnings::register;
if (warnings::enabled()) {
 warnings::warn("some warning");
}

if (warnings::enabled("void")) {
 warnings::warn("void", "some warning");
}

if (warnings::enabled($object)) {
 warnings::warn($object, "some warning");
}

warnings::warnif("some warning");
warnings::warnif("void", "some warning");
warnings::warnif($object, "some warning");
```

**DESCRIPTION**

If no import list is supplied, all possible warnings are either enabled or disabled.

A number of functions are provided to assist module authors.

**use warnings::register**

Creates a new warnings category with the same name as the package where the call to the pragma is used.

**warnings::enabled()**

Use the warnings category with the same name as the current package.

Return TRUE if that warnings category is enabled in the calling module. Otherwise returns FALSE.

**warnings::enabled(\$category)**

Return TRUE if the warnings category, *\$category*, is enabled in the calling module. Otherwise returns FALSE.

**warnings::enabled(\$object)**

Use the name of the class for the object reference, *\$object*, as the warnings category.

Return TRUE if that warnings category is enabled in the first scope where the object is used. Otherwise returns FALSE.

**warnings::warn(\$message)**

Print *\$message* to STDERR.

Use the warnings category with the same name as the current package.

If that warnings category has been set to "FATAL" in the calling module then die. Otherwise return.

**warnings::warn(\$category, \$message)**

Print *\$message* to STDERR.



If the warnings category, `$category`, has been set to "FATAL" in the calling module then die. Otherwise return.

`warnings::warn($object, $message)`

Print `$message` to STDERR.

Use the name of the class for the object reference, `$object`, as the warnings category.

If that warnings category has been set to "FATAL" in the scope where `$object` is first used then die. Otherwise return.

`warnings::warnif($message)`

Equivalent to:

```
if (warnings::enabled())
{ warnings::warn($message) }
```

`warnings::warnif($category, $message)`

Equivalent to:

```
if (warnings::enabled($category))
{ warnings::warn($category, $message) }
```

`warnings::warnif($object, $message)`

Equivalent to:

```
if (warnings::enabled($object))
{ warnings::warn($object, $message) }
```

See [Pragmatic Modules](#) and [perllexwarn](#).

**NAME**

warnings::register – warnings import function

**SYNOPSIS**

```
use warnings::register ;
```

**DESCRIPTION**

Create a warnings category with the same name as the current package.

See [Pragmatic Modules](#) and [perllexwarn](#).

**NAME**

Win32 – Interfaces to some Win32 API Functions

**DESCRIPTION**

Perl on Win32 contains several functions to access Win32 APIs. Some are included in Perl itself (on Win32) and some are only available after explicitly requesting the Win32 module with:

```
use Win32;
```

The builtin functions are marked as [CORE] and the other ones as [EXT] in the following alphabetical listing. The Win32 module is not part of the Perl source distribution; it is distributed in the libwin32 bundle of Win32::\* modules on CPAN. The module is already preinstalled in binary distributions like ActivePerl.

**Alphabetical Listing of Win32 Functions**

Win32::AbortSystemShutdown(MACHINE)

[EXT] Aborts a system shutdown (started by the InitiateSystemShutdown function) on the specified MACHINE.

Win32::BuildNumber()

[CORE] Returns the ActivePerl build number. This function is only available in the ActivePerl binary distribution.

Win32::CopyFile(FROM, TO, OVERWRITE)

[CORE] The Win32::CopyFile() function copies an existing file to a new file. All file information like creation time and file attributes will be copied to the new file. However it will **not** copy the security information. If the destination file already exists it will only be overwritten when the OVERWRITE parameter is true. But even this will not overwrite a read-only file; you have to unlink() it first yourself.

Win32::DomainName()

[CORE] Returns the name of the Microsoft Network domain that the owner of the current perl process is logged into. This function does **not** work on Windows 9x.

Win32::ExpandEnvironmentStrings(STRING)

[EXT] Takes STRING and replaces all referenced environment variable names with their defined values. References to environment variables take the form %VariableName%. Case is ignored when looking up the VariableName in the environment. If the variable is not found then the original %VariableName% text is retained. Has the same effect as the following:

```
$string =~ s/%([^\%]*)%/$ENV{$1} || "%$1%"/eg
```

Win32::FormatMessage(ERRORCODE)

[CORE] Converts the supplied Win32 error number (e.g. returned by Win32::GetLastError()) to a descriptive string. Analogous to the perror() standard-C library function. Note that \$^E used in a string context has much the same effect.

```
C:\> perl -e "$^E = 26; print $^E;"
The specified disk or diskette cannot be accessed
```

Win32::FsType()

[CORE] Returns the name of the filesystem of the currently active drive (like 'FAT' or 'NTFS'). In list context it returns three values: (FSTYPE, FLAGS, MAXCOMPLEN). FSTYPE is the filesystem type as before. FLAGS is a combination of values of the following table:

|            |                                   |
|------------|-----------------------------------|
| 0x00000001 | supports case-sensitive filenames |
| 0x00000002 | preserves the case of filenames   |
| 0x00000004 | supports Unicode in filenames     |
| 0x00000008 | preserves and enforces ACLs       |

|            |                                           |
|------------|-------------------------------------------|
| 0x00000010 | supports file-based compression           |
| 0x00000020 | supports disk quotas                      |
| 0x00000040 | supports sparse files                     |
| 0x00000080 | supports reparse points                   |
| 0x00000100 | supports remote storage                   |
| 0x00008000 | is a compressed volume (e.g. DoubleSpace) |
| 0x00010000 | supports object identifiers               |
| 0x00020000 | supports the Encrypted File System (EFS)  |

MAXCOMPLEN is the maximum length of a filename component (the part between two backslashes) on this file system.

### Win32::FreeLibrary(HANDLE)

[EXT] Unloads a previously loaded dynamic-link library. The HANDLE is no longer valid after this call. See [LoadLibrary/Win32::LoadLibrary\(LIBNAME\)](#) for information on dynamically loading a library.

### Win32::GetArchName()

[EXT] Use of this function is deprecated. It is equivalent with `$ENV{PROCESSOR_ARCHITECTURE}`. This might not work on Win9X.

### Win32::GetChipName()

[EXT] Returns the processor type: 386, 486 or 586 for Intel processors, 21064 for the Alpha chip.

### Win32::GetCwd()

[CORE] Returns the current active drive and directory. This function does not return a UNC path, since the functionality required for such a feature is not available under Windows 95.

### Win32::GetFullPathName(FILENAME)

[CORE] GetFullPathName combines the FILENAME with the current drive and directory name and returns a fully qualified (aka, absolute) path name. In list context it returns two elements: (PATH, FILE) where PATH is the complete pathname component (including trailing backslash) and FILE is just the filename part. Note that no attempt is made to convert 8.3 components in the supplied FILENAME to longnames or vice-versa. Compare with Win32::GetShortPathName and Win32::GetLongPathName.

This function has been added for Perl 5.6.

### Win32::GetLastError()

[CORE] Returns the last error value generated by a call to a Win32 API function. Note that `$_E` used in a numeric context amounts to the same value.

### Win32::GetLongPathName(PATHNAME)

[CORE] Returns a representation of PATHNAME composed of longname components (if any). The result may not necessarily be longer than PATHNAME. No attempt is made to convert PATHNAME to the absolute path. Compare with Win32::GetShortPathName and Win32::GetFullPathName.

This function has been added for Perl 5.6.

### Win32::GetNextAvailDrive()

[CORE] Returns a string in the form of "<d:" where <d is the first available drive letter.

### Win32::GetOSVersion()

[CORE] Returns the array (STRING, MAJOR, MINOR, BUILD, ID), where the elements are, respectively: An arbitrary descriptive string, the major version number of the operating system, the minor version number, the build number, and a digit indicating the actual operating system. For the ID, the values are 0 for Win32s, 1 for Windows 9X and 2 for Windows NT/2000/XP. In scalar context it returns just the ID.

Currently known values for ID MAJOR and MINOR are as follows:

| OS                  | ID | MAJOR | MINOR |
|---------------------|----|-------|-------|
| Win32s              | 0  | -     | -     |
| Windows 95          | 1  | 4     | 0     |
| Windows 98          | 1  | 4     | 10    |
| Windows Me          | 1  | 4     | 90    |
| Windows NT 3.51     | 2  | 3     | 51    |
| Windows NT 4        | 2  | 4     | 0     |
| Windows 2000        | 2  | 5     | 0     |
| Windows XP          | 2  | 5     | 1     |
| Windows .NET Server | 2  | 5     | 1     |

Unfortunately as of June 2002 there is no way to distinguish between .NET servers and XP servers without using additional modules.

`Win32::GetOSName()`

[EXT] In scalar context returns the name of the Win32 operating system being used. In list context returns a two element list of the OS name and whatever edition information is known about the particular build (for Win9x boxes) and whatever service packs have been installed. The latter is roughly equivalent to the first item returned by `GetOSVersion()` in list context.

Currently the possible values for the OS name are

`Win32s Win95 Win98 WinMe Win2000 WinXP/.Net WinNT3.51 WinNT4`

This routine is just a simple interface into `GetOSVersion()`. More specific or demanding situations should use that instead. Another option would be to use `POSIX::uname()`, however the latter appears to report only the OS family name and not the specific OS. In scalar context it returns just the ID.

`Win32::GetShortPathName(PATHNAME)`

[CORE] Returns a representation of `PATHNAME` composed only of short (8.3) path components. The result may not necessarily be shorter than `PATHNAME`. Compare with `Win32::GetFullPathName` and `Win32::GetLongPathName`.

`Win32::GetProcAddress(INSTANCE, PROCNAME)`

[EXT] Returns the address of a function inside a loaded library. The information about what you can do with this address has been lost in the mist of time. Use the `Win32::API` module instead of this deprecated function.

`Win32::GetTickCount()`

[CORE] Returns the number of milliseconds elapsed since the last system boot. Resolution is limited to system timer ticks (about 10ms on WinNT and 55ms on Win9X).

`Win32::InitiateSystemShutdown`

(`MACHINE`, `MESSAGE`, `TIMEOUT`, `FORCECLOSE`, `REBOOT`)

[EXT] Shutdown the specified `MACHINE`, notifying users with the supplied `MESSAGE`, within the specified `TIMEOUT` interval. Forces closing of all documents without prompting the user if `FORCECLOSE` is true, and reboots the machine if `REBOOT` is true. This function works only on WinNT.

`Win32::IsWinNT()`

[CORE] Returns non zero if the Win32 subsystem is Windows NT.

`Win32::IsWin95()`

[CORE] Returns non zero if the Win32 subsystem is Windows 95.

**Win32::LoadLibrary(LIBNAME)**

[EXT] Loads a dynamic link library into memory and returns its module handle. This handle can be used with Win32::GetProcAddress and Win32::FreeLibrary. This function is deprecated. Use the Win32::API module instead.

**Win32::LoginName()**

[CORE] Returns the username of the owner of the current perl process.

**Win32::LookupAccountName(SYSTEM, ACCOUNT, DOMAIN, SID, SIDTYPE)**

[EXT] Looks up ACCOUNT on SYSTEM and returns the domain name the SID and the SID type.

**Win32::LookupAccountSID(SYSTEM, SID, ACCOUNT, DOMAIN, SIDTYPE)**

[EXT] Looks up SID on SYSTEM and returns the account name, domain name, and the SID type.

**Win32::MsgBox(MESSAGE [, FLAGS [, TITLE]])**

[EXT] Create a dialogbox containing MESSAGE. FLAGS specifies the required icon and buttons according to the following table:

|                    |   |                                       |
|--------------------|---|---------------------------------------|
| 0                  | = | OK                                    |
| 1                  | = | OK and Cancel                         |
| 2                  | = | Abort, Retry, and Ignore              |
| 3                  | = | Yes, No and Cancel                    |
| 4                  | = | Yes and No                            |
| 5                  | = | Retry and Cancel                      |
| MB_ICONSTOP        |   | "X" in a red circle                   |
| MB_ICONQUESTION    |   | question mark in a bubble             |
| MB_ICONEXCLAMATION |   | exclamation mark in a yellow triangle |
| MB_ICONINFORMATION |   | "i" in a bubble                       |

TITLE specifies an optional window title. The default is "Perl".

The function returns the menu id of the selected push button:

|   |        |
|---|--------|
| 0 | Error  |
| 1 | OK     |
| 2 | Cancel |
| 3 | Abort  |
| 4 | Retry  |
| 5 | Ignore |
| 6 | Yes    |
| 7 | No     |

**Win32::NodeName()**

[CORE] Returns the Microsoft Network node-name of the current machine.

**Win32::RegisterServer(LIBRARYNAME)**

[EXT] Loads the DLL LIBRARYNAME and calls the function DllRegisterServer.

**Win32::SetChildShowWindow(SHOWWINDOW)**

[CORE] Sets the *ShowMode* of child processes started by `system()`. By default `system()` will create a new console window for child processes if Perl itself is not running from a console. Calling `SetChildShowWindow(0)` will make these new console windows invisible. Calling `SetChildShowWindow()` without arguments reverts `system()` to the default behavior. The return value of `SetChildShowWindow()` is the previous setting or undef.

[EXT] The following symbolic constants for SHOWWINDOW are available (but not exported) from the Win32 module: SW\_HIDE, SW\_SHOWNORMAL, SW\_SHOWMINIMIZED,

SW\_SHOWMAXIMIZED and SW\_SHOWNOACTIVATE.

Win32::SetCwd(NEWDIRECTORY)

[CORE] Sets the current active drive and directory. This function does not work with UNC paths, since the functionality required to required for such a feature is not available under Windows 95.

Win32::SetLastError(ERROR)

[CORE] Sets the value of the last error encountered to ERROR. This is that value that will be returned by the `Win32::GetLastError()` function. This functions has been added for Perl 5.6.

Win32::Sleep(TIME)

[CORE] Pauses for TIME milliseconds. The timeslices are made available to other processes and threads.

Win32::Spawn(COMMAND, ARGS, PID)

[CORE] Spawns a new process using the supplied COMMAND, passing in arguments in the string ARGS. The pid of the new process is stored in PID. This function is deprecated. Please use the Win32::Process module instead.

Win32::UnregisterServer(LIBRARYNAME)

[EXT] Loads the DLL LIBRARYNAME and calls the function DllUnregisterServer.

**NAME**

GDBM\_File – Perl5 access to the gdbm library.

**SYNOPSIS**

```
use GDBM_File ;
tie %hash, 'GDBM_File', $filename, &GDBM_WRCREAT, 0640;
Use the %hash array.
untie %hash ;
```

**DESCRIPTION**

**GDBM\_File** is a module which allows Perl programs to make use of the facilities provided by the GNU gdbm library. If you intend to use this module you should really have a copy of the gdbm manualpage at hand.

Most of the libgdbm.a functions are available through the GDBM\_File interface.

**AVAILABILITY**

gdbm is available from any GNU archive. The master site is <ftp.gnu.org>, but you are strongly urged to use one of the many mirrors. You can obtain a list of mirror sites from <http://www.gnu.org/order/ftp.html>.

**BUGS**

The available functions and the gdbm/perl interface need to be documented.

**SEE ALSO**

*[perl\(1\)](#), [DB\\_File\(3\)](#), [perldbmsfilter](#).*



## NAME

ODBM\_File – Tied access to odbm files

## SYNOPSIS

```
use Fcntl; # For O_RDWR, O_CREAT, etc.
use ODBM_File;

Now read and change the hash
$h{newkey} = newvalue;
print $h{oldkey};
...

untie %h;
```

## DESCRIPTION

ODBM\_File establishes a connection between a Perl hash variable and a file in ODBM\_File format. You can manipulate the data in the file just as if it were in a Perl hash, but when your program exits, the data will remain in the file, to be used the next time your program runs.

Use ODBM\_File with the Perl built-in `tie` function to establish the connection between the variable and the file. The arguments to `tie` should be:

1. The hash variable you want to tie.
2. The string "ODBM\_File". (This tells Perl to use the ODBM\_File package to perform the functions of the hash.)
3. The name of the file you want to tie to the hash.
4. Flags. Use one of:

O\_RDONLY

Read-only access to the data in the file.

O\_WRONLY

Write-only access to the data in the file.

O\_RDWR

Both read and write access.

If you want to create the file if it does not exist, add O\_CREAT to any of these, as in the example. If you omit O\_CREAT and the file does not already exist, the `tie` call will fail.

5. The default permissions to use if a new file is created. The actual permissions will be modified by the user's `umask`, so you should probably use 0666 here. (See [umask](#).)

## DIAGNOSTICS

On failure, the `tie` call returns an undefined value and probably sets `$!` to contain the reason the file could not be tied.

**odbm store returned -1, errno 22, key "... at ...**

This warning is emitted when you try to store a key or a value that is too long. It means that the change was not recorded in the database. See BUGS AND WARNINGS below.

## BUGS AND WARNINGS

There are a number of limits on the size of the data that you can store in the ODBM file. The most important is that the length of a key, plus the length of its associated value, may not exceed 1008 bytes.

See [tie](#), [perlodbmfilter](#), [Fcntl](#)

## NAME

NDBM\_File – Tied access to ndbm files

## SYNOPSIS

```
use Fcntl; # For O_RDWR, O_CREAT, etc.
use NDBM_File;

Now read and change the hash
$h{newkey} = newvalue;
print $h{oldkey};
...

untie %h;
```

## DESCRIPTION

NDBM\_File establishes a connection between a Perl hash variable and a file in NDBM\_File format. You can manipulate the data in the file just as if it were in a Perl hash, but when your program exits, the data will remain in the file, to be used the next time your program runs.

Use NDBM\_File with the Perl built-in `tie` function to establish the connection between the variable and the file. The arguments to `tie` should be:

1. The hash variable you want to tie.
2. The string "NDBM\_File". (This tells Perl to use the NDBM\_File package to perform the functions of the hash.)
3. The name of the file you want to tie to the hash.
4. Flags. Use one of:

O\_RDONLY

Read-only access to the data in the file.

O\_WRONLY

Write-only access to the data in the file.

O\_RDWR

Both read and write access.

If you want to create the file if it does not exist, add O\_CREAT to any of these, as in the example. If you omit O\_CREAT and the file does not already exist, the `tie` call will fail.

5. The default permissions to use if a new file is created. The actual permissions will be modified by the user's `umask`, so you should probably use 0666 here. (See [umask](#).)

## DIAGNOSTICS

On failure, the `tie` call returns an undefined value and probably sets `$!` to contain the reason the file could not be tied.

**ndbm store returned -1, errno 22, key "... " at ...**

This warning is emitted when you try to store a key or a value that is too long. It means that the change was not recorded in the database. See **BUGS AND WARNINGS** below.

## BUGS AND WARNINGS

There are a number of limits on the size of the data that you can store in the NDBM file. The most important is that the length of a key, plus the length of its associated value, may not exceed 1008 bytes.

See [tie](#), [perlndbmfilter](#), [Fcntl](#)

## NAME

SDBM\_File – Tied access to sdbm files

## SYNOPSIS

```
use Fcntl; # For O_RDWR, O_CREAT, etc.
use SDBM_File;

tie(%h, 'SDBM_File', 'filename', O_RDWR|O_CREAT, 0666)
 or die "Couldn't tie SDBM file 'filename': $!; aborting";

Now read and change the hash
$h{newkey} = newvalue;
print $h{oldkey};
...

untie %h;
```

## DESCRIPTION

SDBM\_File establishes a connection between a Perl hash variable and a file in SDBM\_File format. You can manipulate the data in the file just as if it were in a Perl hash, but when your program exits, the data will remain in the file, to be used the next time your program runs.

Use SDBM\_File with the Perl built-in `tie` function to establish the connection between the variable and the file. The arguments to `tie` should be:

1. The hash variable you want to tie.
2. The string "SDBM\_File". (This tells Perl to use the SDBM\_File package to perform the functions of the hash.)
3. The name of the file you want to tie to the hash.
4. Flags. Use one of:

`O_RDONLY`

Read-only access to the data in the file.

`O_WRONLY`

Write-only access to the data in the file.

`O_RDWR`

Both read and write access.

If you want to create the file if it does not exist, add `O_CREAT` to any of these, as in the example. If you omit `O_CREAT` and the file does not already exist, the `tie` call will fail.

5. The default permissions to use if a new file is created. The actual permissions will be modified by the user's `umask`, so you should probably use 0666 here. (See [umask](#).)

## DIAGNOSTICS

On failure, the `tie` call returns an undefined value and probably sets `$!` to contain the reason the file could not be tied.

**sdbm store returned -1, errno 22, key "..."** at ...

This warning is emitted when you try to store a key or a value that is too long. It means that the change was not recorded in the database. See **BUGS AND WARNINGS** below.

## BUGS AND WARNINGS

There are a number of limits on the size of the data that you can store in the SDBM file. The most important is that the length of a key, plus the length of its associated value, may not exceed 1008 bytes.

See *[tie](#)*, *[perldbmfilter](#)*, *[Fcntl](#)*

**NAME**

XS::Typemap – module to test the XS typemaps distributed with perl

**SYNOPSIS**

```
use XS::Typemap;

$output = T_IV($input);
$output = T_PV($input);
@output = T_ARRAY(@input);
```

**DESCRIPTION**

This module is used to test that the XS typemaps distributed with perl are working as advertised. A function is available for each typemap definition (eventually). In general each function takes a variable, processes it through the OUTPUT typemap and then returns it using the INPUT typemap.

A test script can then compare the input and output to make sure they are the expected values. When only an input or output function is provided the function will be named after the typemap entry and have either ‘\_IN’ or ‘\_OUT’ appended.

All the functions are exported. There is no reason not to do this since the entire purpose is for testing Perl. Namespace pollution will be limited to the test script.

**NOTES**

This module is for testing only and should not normally be installed.

**AUTHOR**

Tim Jenness <t.jenness@jach.hawaii.edu>

Copyright (C) 2001 Tim Jenness All Rights Reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

**NAME**

JNI – Perl encapsulation of the Java Native Interface

**SYNOPSIS**

```
use JNI;
```

**DESCRIPTION****Exported constants**

```
JNI_ABORT
JNI_COMMIT
JNI_ERR
JNI_FALSE
JNI_H
JNI_OK
JNI_TRUE
```

**AUTHOR**

Copyright 1998, O'Reilly & Associates, Inc.

This package may be copied under the same terms as Perl itself.

**SEE ALSO**

perl(1).

**NAME**

Tutorial – Perl and Java

**SYNOPSIS**

Java and Perl have different strengths and complement each other well.

You can connect them at runtime with tools such as JPL, PJC, or ActiveX. In theory, you can convert Perl to Java bytecode, and vice-versa.

**Note:**

Not actually a conversion.

At this stage, we are generating Java opcodes by walking Perl's syntax tree. This is very different from converting Perl to Java. It's a lot easier!

**1.1 Perl and Java, Compared**

Perl offers rich text processing features, high-level network APIs, excellent database integration, and a centralized repository of reusable code:

- Regular expression engine is a powerful sub language that can perform complex text manipulations and extract data.
- Packages such as libwww-perl (LWP) and libnet are powerful, high-level interfaces to network functionality.
- The Perl DBI is an interface to SQL data sources.
- CPAN provides a centralized, organized archive of reusable code.

Java has a powerful graphical API, has numerous embedded implementations, excellent database integration, but no single recognized repository of reusable code.

- The Swing (JFC) toolkit is a powerful toolkit for developing user interfaces. Java also boasts 2D and 3D graphics APIs.
- Java comes in embedded flavors, such as:
  - Kaffe <http://www.transvirtual.com/> – embedded implementations for different platforms
  - Waba <http://www.wabasoft.com/> – a subset of Java for Windows CE and PalmOS
  - It's embedded into web browsers (Netscape and MS Internet Explorer)
  - and more...
- Java's JDBC is similar to Perl's DBI
- Java has many different repositories of code. Efforts such as the Giant Java Tree <http://www.gjt.org/> attempt to create a unified repository.

**1.2 Opportunities to Combine Java and Perl**

You have a Java program with a lot of data that needs to be parsed, filed, briefed, debriefed, and numbered.

You want to build your GUI in Java, but let Perl do the heavy lifting.

You've adopted the "Java is a systems language, Perl is a scripting language" paradigm, and it works for you.

You're not sure which regex implementation to use:

```
org.teeth.green.loony.raving.monster.regex.*;
com.zeppelin.regex.*;
```

You want the **best of both worlds**.

### 1.3 Important Differences between Java and Perl

- `perl` compiles and executes programs each time you run them (unless you use the Perl compiler).
- `javac` compiles programs in advance, `java` runs them in the Java interpreter.
- The Java interpreter supports method overloading (methods can have the same name, but are differentiated on the basis of their argument types). Overloaded methods generally perform the same function, but methods with a shorter argument list often use defaults:

```
// Draw a circle in the center of the screen
int drawCircle(int radius);
```

```
// Draw a circle at specified coordinates
int drawCircle(int radius, int h, int k);
```

- The Perl interpreter doesn't support method overloading. In JPL, when we call Java from Perl, we need to use some tricks to specify the Java method we want to invoke. We'll learn about this when we see JPL's `getmeth` function.

#### Note:

At the time this presentation was prepared, JPL did not work with Perl for Win32. However, JPL is in the core Perl distribution, and there are plans to make it work with Perl for Win32.

With that in mind, I'm presenting the JPL material first, because it is of interest to both Win32 and Unix Perl people. The Win32-specific stuff (alternatives to JPL) will come last. I won't be offended if the Unix people leave when I move to this section of the tutorial, since there is no Unix material in that section. I'm perfectly happy to take questions between JPL and ActiveX sections.

A subset of JPL now works on Win32. You can embed Java in Perl, but you cannot embed Perl in Java (yet).

## 2.1 JPL Overview

Let's look at an overview of JPL.

### 2.1.1 Calling Perl from Java

Well-supported by JPL, but it is a complicated process:

- The JPL preprocessor parses the *.jpl* file and generates C code wrappers for Perl methods. It also generates Java and Perl source files.
- The C compiler compiles the wrapper and links it to the *libPerlInterpreter.so* shared library, producing a shared library for the wrapper.
- The Java compiler compiles the Java source file, which uses native methods to load the wrapper.
- The wrapper connects the Java code to the Perl code in the Perl source file.

Fortunately, a generic ***Makefile.PL*** simplifies the process. This is a Perl script that generates a *Makefile* for you.

### 2.1.2 Calling Java from Perl

This works best when Perl is embedded within a Java program.

The JNI Perl module creates and loads a JVM. There is no precompiler, nothing extra — it's just a Perl module and extension.

B<A Problem, Though>. In theory, you can call Java from standalone Perl programs, but this doesn't work because some implementations of Java use a user-level threads package (green threads) that override some functions in the C library. Perl is comfortable using these functions, but Java is not happy using the standard C



library functions.

So, with green threads, you can't reliably embed Java in a standalone Perl program.

Many Java implementations now use native threads. JPL has been tested on Solaris with JDK 1.1.x and native threads, but not on Linux.

**Note:**

Oddly enough, this is the only way it works on Win32.

On Unix, I've still had trouble, even with native threads. I might need to recompile perl with `-DREENTRANT`, but I'm not sure.

## 2.2 Working with JPL

How to set up a JPL application, compile, and install it.

### 2.2.1 Setting up a Project

- 1 The `install-jpl` script creates the `setvars` script. Source the output of `setvars` into your shell when you want to develop or run JPL applications.
- 2 Create a directory with the name of your project, such as *Frotz*. (if you want to use the generic *Makefile.PL*, you need a separate directory for each JPL class you create).
- 3 Copy the generic *Makefile.PL* into the project directory. The `jpl/Sample` directory in the Perl distribution includes the generic *Makefile.PL*.
- 4 Write a `.jpl` program with the same name as the project (such as *Frotz.jpl*)

### 2.2.2 Compiling and Installing a Project

Type `make` to compile the application, and `make install` to install it. This installs the application in the `jpl` directory you created when you installed JPL.

B<Beware>. The default `I<jpl>` directory is the same as the directory you install it `I<from>`. If you go with the default and delete your Perl source, you'll delete your JPL installation!

Type `java Frotz` (or the name you chose in step 2 of section 2.2.1) to run it

### 2.2.3 What's in the `jpl` Directory?

- **libPerlInterpreter.so**: a shared library that loads the Perl interpreter.
- Compiled *.class* files for JPL applications you have written.
- Native code shared library wrappers for JPL applications you have written.
- Perl scripts that contain the Perl code to load at runtime.

Beware. If you issue the `C<make>` command and then run the examples in your development directory, you might be in for a surprise! If the JPL directories come first in your `CLASSPATH` and `LD_LIBRARY_PATH`, you'll keep running the installed, older version, rather than the one you are developing

**Note:**

"Source" means to load it into your current shell, with something like:

```
eval-backtick-setvars-backtick
```

as opposed to just executing it, because then only the subshell gets the environment vars.

## 2.3 Calling Perl from Java

Now, we'll look at how you can invoke Perl from Java.

### 2.3.1 Perl Methods

You can put Perl methods in your *.jpl* file. Perl methods are declared `perl` and use double curly braces to make life easier on the JPL preprocessor:

```
perl int perlMultiply(int a, int b) {{
 my $result = $a * $b;
 return $result;
}}
```

In your Java code, you can invoke Perl methods like a Java method. The native code wrappers take care of running the Perl code:

```
public void invokePerlFunction() {
 int x = 3;
 int y = 6;
 int retval = perlMultiply(x, y);
 System.out.println(x + " * " + y + " = " + retval);
}

class MethodDemo

class MethodDemo {
 // A Perl method to multiply two numbers and
 // return the result.
 //
 perl int perlMultiply(int a, int b) {{
 my $result = $a * $b;
 return $result;
 }}

 // A Java method to call the Perl function.
 //
 public void invokePerlFunction() {
 int x = 3;
 int y = 6;
 int retval = perlMultiply(x, y);
 System.out.println(x + " * " + y + " = " + retval);
 }

 public static void main(String[] args) {
 MethodDemo demo = new MethodDemo();
 demo.invokePerlFunction();
 }
}
```

### Where did `$self` go?

Don't worry, `$self` is still there. JPL takes care of fetching it, as well as all the other arguments:

```
perl int perlMultiply(int a, int b) {{
 my $result = $a * $b;
 return $result;
}}

perl void calculateProduct() {{
 my $x = 3;
 my $y = 6;
```

```

 my $retval = $self->perlMultiply($x, $y);
 print "$x * $y = $retval\n";
 }}

```

B<Note>. JPL takes care of putting all the arguments, including C<\$self>, into variables. If you see a variable in the function header, you will get a variable of the same name without having to use C<shift> or C<@\_>, guaranteed.

NOTE: I've added a line that prints the output of "ref dollar sign self" You'll see this when I run the demo.

```

class SelfDemo {
 // A Perl method to multiply two values.
 //
 perl int perlMultiply(int a, int b) {{
 my $result = $a * $b;
 return $result;
 }}

 // A Perl method to invoke another Perl method.
 //
 perl void calculateProduct() {{
 my $x = 3;
 my $y = 6;
 # Ahhh. There's our old friend, $self!
 #
 my $retval = $self->perlMultiply($x, $y);
 # Display the results.
 #
 print "$x * $y = $retval\n";
 }}

 public static void main(String[] args) {
 SelfDemo demo = new SelfDemo();
 demo.calculateProduct();
 }
}

```

## Passing Arrays

If you pass an array from Java into a Perl method, it arrives in the form of a scalar reference.

Use the `GetIntArrayElements()` JNI function to convert that scalar into an array of integers.

```

perl void min_max(int[] data) {{
 # Get the array elements
 #
 my @new_array = GetIntArrayElements($data);
 # Sort the array numerically
 #
 my @sorted = sort { $a <=> $b } @new_array;

 print "Min: $sorted[0], ",
 "Max: $sorted[$#sorted]\n";
}}

void minMaxDemo() {
 int[] data = {101, 99, 42, 666, 23};
 min_max(data);
}

```

```
}
```

Some JNI Array Functions

**GetBooleanArrayElements( scalar )**

Converts scalar to an array of booleans.

**GetByteArrayElements( scalar )**

Converts scalar to an array of bytes.

**GetCharArrayElements( scalar )**

Converts scalar to an array of characters.

**GetShortArrayElements( scalar )**

Converts scalar to an array of short integers.

**GetIntArrayElements( scalar )**

Converts scalar to an array of integers.

**GetLongArrayElements( scalar )**

Converts scalar to an array of long integers.

**GetFloatArrayElements( scalar )**

Converts scalar to an array of floating point numbers.

**GetDoubleArrayElements( scalar )**

Converts scalar to an array of double precision numbers.

**GetArrayLength( scalar )**

Returns the length of the array.

**PerlTakesArray.jpl**

```
// Show how to pass an array from Java to Perl.
```

```
//
```

```
public class PerlTakesArray {
perl void min_max(int[] data) {{
 # Get the array elements
 #
 my @new_array = GetIntArrayElements($data);
 # Sort the array numerically
 #
 my @sorted = sort {$a <=> $b} @new_array;
 print "Min: $sorted[0], ",
 "Max: $sorted[$#sorted]\n";
}}

void minMaxDemo() {
 // Create an array and ask Perl to tell us
 // the min and max values.
 int[] data = {101, 99, 42, 666, 23};
 min_max(data);
}

public static void main(String[] argv) {
 PerlTakesArray demo = new PerlTakesArray();
 demo.minMaxDemo();
}
```

```
}
```

### 2.3.4 Passing Arrays of Objects

Working with arrays of objects is a little more complicated, because you need to work with them one at a time.

Fetch one element at a time with `GetObjectArrayElement()`, which returns an object of type `java.lang.Object` (the most generic type).

Explicitly cast the `Object` to its real type with `bless()`.

```
perl void sortArray(String[] names) {{
 my @new_array;
 for (my $i = 0; $i < GetArrayLength($names); $i++) {
 my $string = GetObjectArrayElement($names, $i);
 bless $string, "java::lang::String";
 push @new_array, $string;
 }
 print join(', ', sort @new_array), "\n";
}}

void arrayDemo() {
 String[] names = {"Omega", "Gamma", "Beta", "Alpha"};
 sortArray(names);
}
```

Note. `String` is not a primitive type: it is a class (`java.lang.String`). So, you need to use this technique for `Strings` as well. You can't use the technique in 2.3.3.

`PerlTakesObjectArray.jpl`

```
public class PerlTakesObjectArray {
 // Perl method to sort an array of strings.
 //
 perl void sortArray(String[] names) {{
 my @new_array; # an array to copy names[] to

 # Fetch each element from the array.
 for (my $i = 0; $i < GetArrayLength($names); $i++) {
 # Get the object (it's not a String yet!) at
 # the current index ($i).
 my $string = GetObjectArrayElement($names, $i);

 # Cast (bless) it into a String.
 bless $string, "java::lang::String";

 # Add it to the array.
 push @new_array, $string;
 }

 # Print the sorted, comma-delimited array.
 print join(', ', sort @new_array), "\n";
 }}

 // Create a String array and ask Perl to sort it for us.
 //

 void arrayDemo() {
 String[] names = {"Omega", "Gamma", "Beta", "Alpha"};
 sortArray(names);
 }
}
```

```

 }

 public static void main(String[] argv) {
 PerlTakesObjectArray demo = new PerlTakesObjectArray();
 demo.arrayDemo();
 }
}

```

### 2.3.5 Returning Arrays from Perl to Java

To write a Perl method that returns an array, declare its return value as an array type. Make sure you return a reference to the array, not a list:

```

perl int[] getTime() {{
 my ($sec, $min, $hour, @unused) = localtime(time);
 # Return an array with seconds, minutes, hours
 my @time_array = ($sec, $min, $hour);
 return \@time_array;
}}

void testArray() {
 int time[] = getTime();
 System.out.println(time[2] + ":" + time[1]);
}

```

PerlGivesArray.jpl

```

// Simple JPL demo to show how to send an array to Java
// from Perl

class PerlGivesArray {
 // Call the Perl method to get an array and print
 // the hour and minute elements.

 void testArray() {
 int time[] = getTime();
 System.out.println(time[2] + ":" + time[1]);
 }

 // Perl method that returns an array reference.
 //
 perl int[] getTime() {{
 # Get the first three arguments from localtime,
 # discard the rest.
 my ($sec, $min, $hour, @unused) = localtime(time);

 # Return an array with seconds, minutes, hours
 my @time_array = ($sec, $min, $hour);
 return \@time_array;
 }}

 public static void main(String[] argv) {
 PerlGivesArray demo = new PerlGivesArray();
 demo.testArray();
 }
}

```

### 2.3.6 Arrays from Strings

JPL will slice Perl strings up into Java arrays for you. If you declare a Perl method as an array type and return a string (instead of an array reference), JPL splits up the elements into an array.

Consider this example, where a GIF stored in a string gets turned into an array of bytes so Java can make an Image out of it:

```
void generateImage() {
 Toolkit kit = Toolkit.getDefaultToolkit();
 byte[] image_data = mkImage();
 img = kit.createImage(image_data);
}

perl byte[] mkImage() {{
 use GD;
 my $im = new GD::Image($self->width, $self->height);
 my $white = $im->colorAllocate(255, 255, 255);
 my $blue = $im->colorAllocate(0, 0, 255);
 $im->fill($white, 0, 0);
 $im->string(gdLargeFont, 10, 10, "Hello, World", $blue);
 return $im->gif;
}}
```

GifDemo.jpl

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

/*
 * A JPL program that demonstrates passing byte arrays
 * between Java and Perl
 */

class GIFDemo extends Canvas {
 Image img;
 int width = 200;
 int height = 30;

 // Constructor for this class.
 public GIFDemo() {
 this.setSize(width, height);
 }

 // Java method to create an image.
 //
 void generateImage() {
 Toolkit kit = Toolkit.getDefaultToolkit();

 // Invoke the mkImage() Perl method to generate an
 // image.

 byte[] image_data = mkImage();

 // Create the image with the byte array we got
 // from the Perl method.

 img = kit.createImage(image_data);
 }

 // A Perl method to generate an image.
 perl byte[] mkImage() {{
```

```

 # Use the GD image manipulation extension.
 use GD;

 # Create a new image with the height and width specified
 # in the enclosing Java class.
 my $im = new GD::Image($self->width, $self->height);

 # Allocate two colors.
 my $white = $im->colorAllocate(255, 255, 255);
 my $blue = $im->colorAllocate(0, 0, 255);

 # Fill the image with white and draw a greeting.
 $im->fill($white, 0, 0);
 $im->string(gdLargeFont, 10, 10,
 "Hello, World", $blue);
 return $im->gif;
}}

// Java uses this to repaint the image when necessary.
public void paint(Graphics g) {
 g.drawImage(img, 0, 0, this);
}

// The entry point.
public static void main(String[] argv) {
 // Set up a frame and create an image.
 Frame f = new Frame("GD Example");
 f.setLayout(new BorderLayout());

 GIFDemo demo = new GIFDemo();
 demo.generateImage();

 f.add("Center", demo);
 f.addWindowListener(new Handler());

 f.pack();
 f.show();
}
}

// A handler to process a request to close a window.
class Handler extends WindowAdapter {
 public void windowClosing(WindowEvent e) {
 System.exit(0);
 }
}
}

```

### 2.3.7 Summary: Calling Perl from Java

- 1 Put your embedded Perl code in methods that are declared `perl`.
- 2 Use double, rather than single, curly braces (`{{` and `}}`).
- 3 Invoke the Perl methods from Java just like any other Java method.



- 4 No need to pull arguments off of @\_ with shift: JPL takes care of this for you. This includes \$self.
- 5 If you pass a Java array into a Perl method, it comes in as a scalar reference.
- 6 Convert references to arrays of primitives with Get\*ArrayElements
- 7 Use GetObjectArrayElement to get elements from arrays of strings and other objects.
- 8 To return an array from a perl method, declare the method as returning an array type, and either:
  - 9 Return an array reference.
  - 10 Return a string: JPL slices it up for you.

## 2.4 Calling Java from Perl

Next, let's look at how to invoke Java from Perl.

### 2.4.1 Java in Perl in Java

Remember the issues from 2.1.2 – this is unstable unless you are calling Java from Perl methods that are themselves embedded in a Java program.

### 2.4.2 Java in Perl: Simple Constructors

Use JPL::Class to load the class:

```
use JPL::Class "java::awt::Frame";
```

Invoke the constructor to create an instance of the class:

```
my $f = java::awt::Frame-new;
```

You've got a reference to a Java object in \$f, a Perl scalar. I think this is cool.

### 2.4.3 Constructors that Take Parameters

If the constructor has parameters, look up the method signature with getmeth:

```
my $new = getmeth("new", ['java.lang.String'], []);
```

The first argument to getmeth is the name of the method. The second argument is a reference to an array that contains a list of the argument types. The final argument to getmeth is a reference to an array containing a single element with the return type. Constructors always have a null (void) return type, even though they return an instance of an object.

Invoke the method through the variable you created:

```
my $f = java::awt::Frame-$new("Frame Demo");
```

Because Java supports method overloading, the only way Java can distinguish between different methods that have the same name is through the method signature. The getmeth function simply returns a mangled, Perl-friendly version of the signature. JPL's AutoLoader takes care of finding the right class.

For example, the method signature for \$new is (Ljava/lang/String;)V. In Perl, this is translated to new\_\_Ljava\_lang\_String\_2\_\_V. Sure, it means something to Java, but thanks to getmeth and JPL's AutoLoader, we don't have to worry about it!

### 2.4.4 More on getmeth

The getmeth function is not just for constructors. You'll use it to look up method signatures for any method that takes arguments.

To use getmeth, just supply the Java names of the types and objects in the argument or return value list. Here are a few examples:

- Two int arguments, void return type:  
`$setSize = getmeth("setSize", ['int', 'int'], []);`
- One argument (java.awt.Component), with a return type of the same:  
`$add = getmeth("add", ['java.awt.Component'],  
 ['java.awt.Component']);`
- Two arguments, a String object and a boolean value, and a void return type:  
`$new = getmeth("new",  
 ['java.lang.String', 'boolean'], []);`
- A String argument with a java.lang.Class return type:  
`$forName = getmeth("forName",  
 ['java.lang.String'],  
 ['java.lang.Class']);`
- No arguments, but a boolean return value:  
`$next = getmeth("next", [], ['boolean']);`

#### 2.4.5 Instance Variables

Java instance variables that belong to a class can be reached through `$self` and a method with the same name as the instance variables:

```
$frame->$setSize($self->width, $self->height);
```

Here is an example:

```
class VarDemo {
 int foo = 100;
 perl int perlChange() {{
 my $current_value = $self->foo;
 # Change foo to ten times itself.
 $self->foo($current_value * 10);
 }}
 void executeChange() {
 perlChange();
 System.out.println(foo);
 }
 public static void main(String[] args) {
 VarDemo demo = new VarDemo();
 demo.executeChange();
 }
}
```

Note. JPL creates these methods with the same name as the variable. You can also supply a value to set the variable's value. If you create a method with this name, it will collide with the one that JPL defines.

FrameDemo.jpl

```
/*
 * FrameDemo - create and show a Frame in Perl.
 *
 */

public class FrameDemo {
 int height = 50;
 int width = 200;
 perl void make_frame () {{
 # Import two Java classes.

 use JPL::Class "java::awt::Frame";
 use JPL::Class "java::awt::Button";

 # Create a Frame and a Button. The two calls to new()
 # have the same signature.

 my $new = getmeth("new", ['java.lang.String'], []);
 my $frame = java::awt::Frame->$new("Frame Demo");
 my $btn = java::awt::Button->$new("Do Not Press Me");

 # Add the button to the frame.

 my $add = getmeth("add", ['java.awt.Component'],
 ['java.awt.Component']);
 $frame->$add($btn);

 # Set the size of the frame and show it.

 my $setSize = getmeth("setSize", ['int', 'int'], []);
 $frame->$setSize($self->width, $self->height);
 $frame->show;
 }}

 public static void main(String[] argv) {
 FrameDemo demo = new FrameDemo();
 demo.make_frame();
 }
}
```

#### 2.4.6 Summary: Calling Java from Perl

- 1 Use JPL::Class to specify a Java class to import.
- 2 You can directly invoke constructors and methods that take no arguments.
- 3 If the constructor or method takes arguments, use getmeth to look up its signature.
- 4 Use \$self to access Java instance variables and methods.

#### COPYRIGHT

Copyright (c) 1999, Brian Jepson

You may distribute this file under the same terms as Perl itself.

Converted from FrameMaker by Kevin Falcone.

**Name**

patching.pod – Appropriate format for patches to the perl source tree

**Where to get this document**

The latest version of this document is available from  
<http://perrin.dimensional.com/perl/perlpatch.html>

**How to contribute to this document**

You may mail corrections, additions, and suggestions to me at [dgris@dimensional.com](mailto:dgris@dimensional.com) but the preferred method would be to follow the instructions set forth in this document and submit a patch 8–).

**Description****Why this document exists**

As an open source project Perl relies on patches and contributions from its users to continue functioning properly and to root out the inevitable bugs. But, some users are unsure as to the *right* way to prepare a patch and end up submitting seriously malformed patches. This makes it very difficult for the current maintainer to integrate said patches into their distribution. This document sets out usage guidelines for patches in an attempt to make everybody's life easier.

**Common problems**

The most common problems appear to be patches being mangled by certain mailers (I won't name names, but most of these seem to be originating on boxes running a certain popular commercial operating system). Other problems include patches not rooted in the appropriate place in the directory structure, and patches not produced using standard utilities (such as diff).

**Proper Patch Guidelines****What to patch**

Generally speaking you should patch the latest development release of perl. The maintainers of the individual branches will see to it that patches are picked up and applied as appropriate.

**How to prepare your patch****Creating your patch**

First, back up the original files. This can't be stressed enough, back everything up *\_first\_*.

Also, please create patches against a clean distribution of the perl source. This ensures that everyone else can apply your patch without clobbering their source tree.

**diff** While individual tastes vary (and are not the point here) patches should be created using either `-u` or `-c` arguments to diff. These produce, respectively, unified diffs (where the changed line appears immediately next to the original) and context diffs (where several lines surrounding the changes are included). See the manpage for diff for more details.

The preferred method for creating a unified diff suitable for feeding to the patch program is:

```
diff -u old-file new-file > patch-file
```

Note the order of files. See below for how to create a patch from two directory trees.

If your patch is for wider consumption, it may be better to create it as a context diff as some machines have broken patch utilities that choke on unified diffs. A context diff is made using `diff -c` rather than `diff -u`.

GNU diff has many desirable features not provided by most vendor-supplied diffs. Some examples using GNU diff:

```
generate a patch for a newly added file
% diff -u /dev/null new/file
```

```
generate a patch to remove a file (patch > v2.4 will remove it cleanly)
% diff -u old/goner /dev/null

get additions, deletions along with everything else, recursively
% diff -ruN olddir newdir

ignore whitespace
% diff -bu a/file b/file

show function name in every hunk (safer, more informative)
% diff -u -F '^[_a-zA-Z0-9]+ *(' old/file new/file
```

### Derived Files

Many files in the distribution are derivative—avoid patching them. Patch the originals instead. Most utilities (like `perldoc`) are in this category, i.e. `patch utils/perldoc.PL` rather than `utils/perldoc`. Similarly, don't create patches for files under `$src_root/ext` from their copies found in `$install_root/lib`. If you are unsure about the proper location of a file that may have gotten copied while building the source distribution, consult the MANIFEST.

### Filenames

The most usual convention when submitting patches for a single file is to make your changes to a copy of the file with the same name as the original. Rename the original file in such a way that it is obvious what is being patched (`$file.dist` or `$file.old` seem to be popular).

If you are submitting patches that affect multiple files then you should backup the entire directory tree (to `$source_root.old/` for example). This will allow `diff -ruN old-dir new-dir` to create all the patches at once.

### Directories

**IMPORTANT:** Patches should be generated from the source root directory, not from the directory that the patched file resides in. This ensures that the maintainer patches the proper file.

For larger patches that are dealing with multiple files or directories, Johan Vromans has written a powerful utility: `makepatch`. See the JV directory on CPAN for the current version. If you have this program available, it is recommended to create a duplicate of the perl directory tree against which you are intending to provide a patch and let `makepatch` figure out all the changes you made to your copy of the sources. As perl comes with a MANIFEST file, you need not delete object files and other derivative files from the two directory trees, `makepatch` is smart about them.

Say, you have created a directory `perl-5.7.1@8685/` for the perl you are taking as the base and a directory `perl-5.7.1@8685-withfoo/` where you have your changes, you would run `makepatch` as follows:

```
makepatch -oldman perl-5.7.1@8685/MANIFEST \
 -newman perl-5.7.1@8685-withfoo/MANIFEST \
 -diff "diff -u" \
 perl-5.7.1@8685 perl-5.7.1@8685-withfoo
```

### Try it yourself

Just to make sure your patch "works", be sure to apply it to the Perl distribution, rebuild everything, and make sure the testsuite runs without incident.

## What to include in your patch

### Description of problem

The first thing you should include is a description of the problem that the patch corrects. If it is a code patch (rather than a documentation patch) you should also include a small test case that illustrates the bug.

### Directions for application

You should include instructions on how to properly apply your patch. These should include the files affected, any shell scripts or commands that need to be run before or after application of the patch, and the command line necessary for application.

### If you have a code patch

If you are submitting a code patch there are several other things that you need to do.

### Comments, Comments, Comments

Be sure to adequately comment your code. While commenting every line is unnecessary, anything that takes advantage of side effects of operators, that creates changes that will be felt outside of the function being patched, or that others may find confusing should be documented. If you are going to err, it is better to err on the side of adding too many comments than too few.

### Style

In general, please follow the particular style of the code you are patching.

In particular, follow these general guidelines for patching Perl sources:

```
8-wide tabs (no exceptions!)
4-wide indents for code, 2-wide indents for nested CPP #defines
try hard not to exceed 79-columns
ANSI C prototypes
uncuddled elses and "K&R" style for indenting control constructs
no C++ style (//) comments, most C compilers will choke on them
mark places that need to be revisited with XXX (and revisit often!)
opening brace lines up with "if" when conditional spans multiple
 lines; should be at end-of-line otherwise
in function definitions, name starts in column 0 (return value is on
 previous line)
single space after keywords that are followed by parens, no space
 between function name and following paren
avoid assignments in conditionals, but if they're unavoidable, use
 extra paren, e.g. "if (a && (b = c)) ..."
"return foo;" rather than "return(foo);"
"if (!foo) ..." rather than "if (foo == FALSE) ..." etc.
```

### Testsuite

When submitting a patch you should make every effort to also include an addition to perl's regression tests to properly exercise your patch. Your testsuite additions should generally follow these guidelines (courtesy of Gurusamy Sarathy <gsar@activestate.com>):

```
Know what you're testing. Read the docs, and the source.
Tend to fail, not succeed.
Interpret results strictly.
Use unrelated features (this will flush out bizarre interactions).
Use non-standard idioms (otherwise you are not testing TIMTOWTDI).
Avoid using hardcoded test numbers whenever possible (the
 EXPECTED/GOT found in t/op/tie.t is much more maintainable,
 and gives better failure reports).
Give meaningful error messages when a test fails.
Avoid using qx// and system() unless you are testing for them. If yo
 do use them, make sure that you cover _all_ perl platforms.
Unlink any temporary files you create.
Promote unforeseen warnings to errors with $SIG{__WARN__}.
Be sure to use the libraries and modules shipped with the version
 being tested, not those that were already installed.
```

Add comments to the code explaining what you are testing for. Make updating the '1..42' string unnecessary. Or make sure that you update it.

Test `_all_` behaviors of a given operator, library, or function:

- All optional arguments
- Return values in various contexts (boolean, scalar, list, lvalue)
- Use both global and lexical variables
- Don't forget the exceptional, pathological cases.

### Test your patch

Apply your patch to a clean distribution, compile, and run the regression test suite (you did remember to add one for your patch, didn't you).

### An example patch creation

This should work for most patches:

```
cp MANIFEST MANIFEST.old
emacs MANIFEST
(make changes)
cd ..
diff -c perl5.7.42/MANIFEST.old perl5.7.42/MANIFEST > mypatch
(testing the patch:)
mv perl5.7.42/MANIFEST perl5.7.42/MANIFEST.new
cp perl5.7.42/MANIFEST.old perl5.7.42/MANIFEST
patch -p < mypatch
(should succeed)
diff perl5.7.42/MANIFEST perl5.7.42/MANIFEST.new
(should produce no output)
```

### Submitting your patch

#### Mailers

Please, please, please (get the point? 8-) don't use a mailer that word wraps your patch. This leaves the patch essentially worthless to the maintainers.

Unfortunately many mailers word wrap the main text of messages, but luckily you can usually send your patches as email attachments without them getting "helpfully" word wrapped.

If you have no choice in mailers and no way to get your hands on a better one, there is, of course, a Perl solution. Just do this:

```
perl -ne 'print pack("u*",$_)' patch > patch.uue
```

and post patch.uue with a note saying to unpack it using

```
perl -ne 'print unpack("u*",$_)' patch.uue > patch
```

#### Subject lines for patches

The subject line on your patch should read

```
[PATCH 5.x.x AREA] Description
```

where the x's are replaced by the appropriate version number. The description should be a very brief but accurate summary of the problem (don't forget this is an email header).

Examples:

```
[PATCH 5.6.4 DOC] fix minor typos
```

```
[PATCH 5.7.9 CORE] New warning for foo() when frobbing
```

```
[PATCH 5.7.16 CONFIG] Added support for fribnatz 1.5
```

The name of the file being patched makes for a poor subject line if no other descriptive text accompanies it.

### Where to send your patch

If your patch is for a specific bug in the Perl core, it should be sent using the `perlbug` utility. Don't forget to describe the problem and the fix adequately.

If it is a patch to a module that you downloaded from CPAN you should submit your patch to that module's author.

If your patch addresses one of the items described in `perltodo.pod`, please discuss your approach **before** you make the patch at `<perl5-porters@perl.org`. Be sure to browse the archives of past discussions (see `perltodo.pod` for archive locations).

## Applying a patch

### General notes on applying patches

The following are some general notes on applying a patch to your perl distribution.

#### `patch -p`

It is generally easier to apply patches with the `-p N` argument to `patch` (where `N` is the number of path components to skip in the files found in the headers). This helps reconcile differing paths between the machine the patch was created on and the machine on which it is being applied.

### Cut and paste

**Never** cut and paste a patch into your editor. This usually clobbers the tabs and confuses `patch`.

### Hand editing patches

Avoid hand editing patches as this almost always screws up the line numbers and offsets in the patch, making it useless.

## Final notes

If you follow these guidelines it will make everybody's life a little easier. You'll have the satisfaction of having contributed to perl, others will have an easy time using your work, and it should be easier for the maintainers to coordinate the occasionally large numbers of patches received.

Also, just because you're not a brilliant coder doesn't mean that you can't contribute. As valuable as code patches are there is always a need for better documentation (especially considering the general level of joy that most programmers feel when forced to sit down and write docs). If all you do is patch the documentation you have still contributed more than the person who sent in an amazing new feature that no one can use because no one understands the code (what I'm getting at is that documentation is both the hardest part to do (because everyone hates doing it) and the most valuable).

Mostly, when contributing patches, imagine that it is **you** receiving hundreds of patches and that it is **your** responsibility to integrate them into the source. Obviously you'd want the patches to be as easy to apply as possible. Keep that in mind. 8-)

## Last Modified

Last modified 21 January 1999 Daniel Grisinger `<dgris@dimensional.com`

## Author and Copyright Information

Copyright (c) 1998, 1999 Daniel Grisinger

Adapted from a posting to `perl5-porters` by Tim Bunce (`Tim.Bunce@ig.co.uk`).

I'd like to thank the `perl5-porters` for their suggestions.



**NAME**

Pumpkin – Notes on handling the Perl Patch Pumpkin And Porting Perl

**SYNOPSIS**

There is no simple synopsis, yet.

**DESCRIPTION**

This document attempts to begin to describe some of the considerations involved in patching, porting, and maintaining perl.

This document is still under construction, and still subject to significant changes. Still, I hope parts of it will be useful, so I'm releasing it even though it's not done.

For the most part, it's a collection of anecdotal information that already assumes some familiarity with the Perl sources. I really need an introductory section that describes the organization of the sources and all the various auxiliary files that are part of the distribution.

**Where Do I Get Perl Sources and Related Material?**

The Comprehensive Perl Archive Network (or CPAN) is the place to go. There are many mirrors, but the easiest thing to use is probably <http://www.cpan.org/README.html>, which automatically points you to a mirror site "close" to you.

**Perl5-porters mailing list**

The mailing list `perl5-porters@perl.org` is the main group working with the development of perl. If you're interested in all the latest developments, you should definitely subscribe. The list is high volume, but generally has a fairly low noise level.

Subscribe by sending the message (in the body of your letter)

```
subscribe perl5-porters
```

to `perl5-porters-request@perl.org`.

Archives of the list are held at:

```
http://www.rosat.mpe-garching.mpg.de/mailling-lists/perl-porters/
```

**How are Perl Releases Numbered?**

Beginning with v5.6.0, even versions will stand for maintenance releases and odd versions for development releases, i.e., v5.6.x for maintenance releases, and v5.7.x for development releases. Before v5.6.0, subversions \_01 through \_49 were reserved for bug-fix maintenance releases, and subversions \_50 through \_99 for unstable development versions.

For example, in v5.6.1, the revision number is 5, the version is 6, and 1 is the subversion.

For compatibility with the older numbering scheme the composite floating point version number continues to be available as the magic variable `$]`, and amounts to `$revision + $version/1000 + $subversion/100000`. This can still be used in comparisons.

```
print "You've got an old perl\n" if $] < 5.005_03;
```

In addition, the version is also available as a string in `$^V`.

```
print "You've got a new perl\n" if $^V and $^V ge v5.6.0;
```

You can also require particular version (or later) with:

```
use 5.006;
```

or using the new syntax available only from v5.6 onward:

```
use v5.6.0;
```

At some point in the future, we may need to decide what to call the next big revision. In the `.package` file used by `metaconfig` to generate `Configure`, there are two variables that might be relevant: `$baserev=5` and `$package=perl5`.

Perl releases produced by the members of `perl5-porters` are usually available on CPAN in the *src/5.0/maint* and *src/5.0/devel* directories.

## Maintenance and Development Subversions

The first rule of maintenance work is "First, do no harm."

Trial releases of bug-fix maintenance releases are announced on `perl5-porters`. Trial releases use the new subversion number (to avoid testers installing it over the previous release) and include a 'local patch' entry in `patchlevel.h`. The distribution file contains the string `MAINT_TRIAL` to make clear that the file is not meant for public consumption.

In general, the names of official distribution files for the public always match the regular expression:

```
^perl\d+\.(\d+)\. \. \d+(-MAINT_TRIAL_\d+)\.tar\.gz$
```

\$1 in the pattern is always an even number for maintenance versions, and odd for developer releases.

In the past it has been observed that pumpkings tend to invent new naming conventions on the fly. If you are a pumpking, before you invent a new name for any of the three types of perl distributions, please inform the guys from the CPAN who are doing indexing and provide the trees of symlinks and the like. They will have to know *in advance* what you decide.

## Why is it called the patch pumpkin?

Chip Salzenberg gets credit for that, with a nod to his cow orker, David Croy. We had passed around various names (baton, token, hot potato) but none caught on. Then, Chip asked:

[begin quote]

Who has the patch pumpkin?

To explain: David Croy once told me once that at a previous job, there was one tape drive and multiple systems that used it for backups. But instead of some high-tech exclusion software, they used a low-tech method to prevent multiple simultaneous backups: a stuffed pumpkin. No one was allowed to make backups unless they had the "backup pumpkin".

[end quote]

The name has stuck.

## Philosophical Issues in Patching and Porting Perl

There are no absolute rules, but there are some general guidelines I have tried to follow as I apply patches to the perl sources. (This section is still under construction.)

### Solve problems as generally as possible

Never implement a specific restricted solution to a problem when you can solve the same problem in a more general, flexible way.

For example, for dynamic loading to work on some SVR4 systems, we had to build a shared `libperl.so` library. In order to build "FAT" binaries on NeXT 4.0 systems, we had to build a special `libperl` library. Rather than continuing to build a contorted nest of special cases, I generalized the process of building `libperl` so that NeXT and SVR4 users could still get their work done, but others could build a shared `libperl` if they wanted to as well.

Contain your changes carefully. Assume nothing about other operating systems, not even closely related ones. Your changes must not affect other platforms.

Spy shamelessly on how similar patching or porting issues have been settled elsewhere.

If feasible, try to keep filenames 8.3-compliant to humor those poor souls that get joy from running Perl

under such dire limitations. There's a script, `check83.pl`, for keeping your nose 8.3-clean.

### Seek consensus on major changes

If you are making big changes, don't do it in secret. Discuss the ideas in advance on `perl5-porters`.

### Keep the documentation up-to-date

If your changes may affect how users use perl, then check to be sure that the documentation is in sync with your changes. Be sure to check all the files `pod/*.pod` and also the *INSTALL* document.

Consider writing the appropriate documentation first and then implementing your change to correspond to the documentation.

### Avoid machine-specific `#ifdef`'s

To the extent reasonable, try to avoid machine-specific `#ifdef`'s in the sources. Instead, use feature-specific `#ifdef`'s. The reason is that the machine-specific `#ifdef`'s may not be valid across major releases of the operating system. Further, the feature-specific tests may help out folks on another platform who have the same problem.

### Machine-specific files

#### source code

If you have many machine-specific `#defines` or `#includes`, consider creating an `osish.h` (`os2ish.h`, `vmsish.h`, and so on) and including that in `perl.h`. If you have several machine-specific files (function emulations, function stubs, build utility wrappers) you may create a separate subdirectory (`djgpp`, `win32`) and put the files in there. Remember to update *MANIFEST* when you add files.

If your system supports dynamic loading but none of the existing methods at `ext/DynaLoader/dl_*.xs` work for you, you must write a new one. Study the existing ones to see what kind of interface you must supply.

#### build hints

There are two kinds of hints: hints for building Perl and hints for extensions. The former live in the `hints` subdirectory, the latter in `ext/*/hints` subdirectories.

The top level hints are Bourne-shell scripts that set, modify and unset appropriate Configure variables, based on the Configure command line options and possibly existing `config.sh` and `Policy.sh` files from previous Configure runs.

The extension hints are written in Perl (by the time they are used `miniperl` has been built) and control the building of their respective extensions. They can be used to for example manipulate compilation and linking flags.

#### build and installation Makefiles, scripts, and so forth

Sometimes you will also need to tweak the Perl build and installation procedure itself, like for example *Makefile.SH* and *installperl*. Tread very carefully, even more than usual. Contain your changes with utmost care.

#### test suite

Many of the tests in `t` subdirectory assume machine-specific things like existence of certain functions, something about filesystem semantics, certain external utilities and their error messages. Use the `$^O` and the `Config` module (which contains the results of the Configure run, in effect the `config.sh` converted to Perl) to either skip (preferably not) or customize (preferable) the tests for your platform.

#### modules

Certain standard modules may need updating if your operating system sports for example a native filesystem naming. You may want to update some or all of the modules `File::Basename`, `File::Spec`, `File::Path`, and `File::Copy` to become aware of your native filesystem syntax and peculiarities.

Remember to have a `$VERSION` in the modules. You can use the `Porting/checkVERSION.pl` script for checking this.

## documentation

If your operating system comes from outside UNIX you almost certainly will have differences in the available operating system functionality (missing system calls, different semantics, whatever). Please document these at *pod/perlport.pod*. If your operating system is the first **not** to have a system call also update the list of "portability-bewares" at the beginning of *pod/perlfunc.pod*.

A file called *README.youros* at the top level that explains things like how to install perl at this platform, where to get any possibly required additional software, and for example what test suite errors to expect, is nice too. Such files are in the process of being written in pod format and will eventually be renamed *INSTALL.youros*.

You may also want to write a separate *.pod* file for your operating system to tell about existing mailing lists, os-specific modules, documentation, whatever. Please name these along the lines of *perlyouros.pod*. [unfinished: where to put this file (the pod/ subdirectory, of course: but more importantly, which/what index files should be updated?)]

## Allow for lots of testing

We should never release a main version without testing it as a subversion first.

## Test popular applications and modules.

We should never release a main version without testing whether or not it breaks various popular modules and applications. A partial list of such things would include majordomo, metaconfig, apache, Tk, CGI, libnet, and libwww, to name just a few. Of course it's quite possible that some of those things will be just plain broken and need to be fixed, but, in general, we ought to try to avoid breaking widely-installed things.

## Automated generation of derivative files

The *embed.h*, *keywords.h*, *opcode.h*, and *perltoc.pod* files are all automatically generated by perl scripts. In general, don't patch these directly; patch the data files instead.

*Configure* and *config\_h.SH* are also automatically generated by *metaconfig*. In general, you should patch the metaconfig units instead of patching these files directly. However, very minor changes to *Configure* may be made in between major sync-ups with the metaconfig units, which tends to be complicated operations. But be careful, this can quickly spiral out of control. Running metaconfig is not really hard.

Also *Makefile* is automatically produced from *Makefile.SH*. In general, look out for all *\*.SH* files.

Finally, the sample files in the *Porting/* subdirectory are generated automatically by the script *U/mksample* included with the metaconfig units. See "*run metaconfig*" below for information on obtaining the metaconfig units.

## How to Make a Distribution

There really ought to be a 'make dist' target, but there isn't. The 'dist' suite of tools also contains a number of tools that I haven't learned how to use yet. Some of them may make this all a bit easier.

Here are the steps I go through to prepare a patch & distribution.

Lots of it could doubtless be automated but isn't. The *Porting/makerel* (make release) perl script does now help automate some parts of it.

## Announce your intentions

First, you should volunteer out loud to take the patch pumpkin. It's generally counter-productive to have multiple people working in secret on the same thing.

At the same time, announce what you plan to do with the patch pumpkin, to allow folks a chance to object or suggest alternatives, or do it for you. Naturally, the patch pumpkin holder ought to incorporate various bug fixes and documentation improvements that are posted while he or she has the pumpkin, but there might also be larger issues at stake.

One of the precepts of the subversion idea is that we shouldn't give the patch pumpkin to anyone unless we have some idea what he or she is going to do with it.

### refresh pod/perltoctoc.pod

Presumably, you have done a full make in your working source directory. Before you make `spotless` (if you do), and if you have changed any documentation in any module or pod file, change to the *pod* directory and run `make toc`.

### run installhtml to check the validity of the pod files

#### update patchlevel.h

Don't be shy about using the subversion number, even for a relatively modest patch. We've never even come close to using all 99 subversions, and it's better to have a distinctive number for your patch. If you need feedback on your patch, go ahead and issue it and promise to incorporate that feedback quickly (e.g. within 1 week) and send out a second patch.

If you update the subversion number, you may need to change the version number near the top of the *Changes* file.

#### run metaconfig

If you need to make changes to `Configure` or `config_h.SH`, it may be best to change the appropriate metaconfig units instead, and regenerate `Configure`.

```
metaconfig -m
```

will regenerate `Configure` and `config_h.SH`. Much more information on obtaining and running metaconfig is in the *U/README* file that comes with Perl's metaconfig units. Perl's metaconfig units should be available on CPAN. A set of units that will work with perl5.005 is in the file *mc\_units-5.005\_00-01.tar.gz* under <http://www.cpan.org/authors/id/ANDYD/>. The *mc\_units* tar file should be unpacked in your main perl source directory. Note: those units were for use with 5.005. There may have been changes since then. Check for later versions or contact [perl5-porters@perl.org](mailto:perl5-porters@perl.org) to obtain a pointer to the current version.

Alternatively, do consider if the *\*ish.h* files might be a better place for your changes.

### MANIFEST

Make sure the MANIFEST is up-to-date. You can use dist's **manicheck** program for this. You can also use

```
perl -w -MExtUtils::Manifest=fullcheck -e fullcheck
```

Both commands will also list extra files in the directory that are not listed in MANIFEST.

The MANIFEST is normally sorted.

If you are using metaconfig to regenerate `Configure`, then you should note that metaconfig actually uses `MANIFEST.new`, so you want to be sure `MANIFEST.new` is up-to-date too. I haven't found the `MANIFEST/MANIFEST.new` distinction particularly useful, but that's probably because I still haven't learned how to use the full suite of tools in the dist distribution.

### Check permissions

All the tests in the `t/` directory ought to be executable. The main makefile used to do a `'chmod t/*/*.*'`, but that resulted in a self-modifying distribution—something some users would strongly prefer to avoid. The *t/TEST* script will check for this and do the `chmod` if needed, but the tests still ought to be executable.

In all, the following files should probably be executable:

```
Configure
configpm
configure.gnu
embed.pl
installperl
installman
keywords.pl
myconfig
```

```
opcode.pl
perly.fixer
t/TEST
t/*/*.t
*.SH
vms/ext/Stdio/test.pl
vms/ext/filespec.t
x2p/*.SH
```

Other things ought to be readable, at least :-).

Probably, the permissions for the files could be encoded in MANIFEST somehow, but I'm reluctant to change MANIFEST itself because that could break old scripts that use MANIFEST.

I seem to recall that some SVR3 systems kept some sort of file that listed permissions for system files; something like that might be appropriate.

## Run Configure

This will build a config.sh and config.h. You can skip this if you haven't changed Configure or config\_h.SH at all. I use the following command

```
sh Configure -Dprefix=/opt/perl -Doptimize=-O -Dusethreads \
-Dcf_by='yourname' \
-Dcf_email='yourname@yourhost.yourplace.com' \
-Dperladmin='yourname@yourhost.yourplace.com' \
-Dmydomain='.yourplace.com' \
-Dmyhostname='yourhost' \
-des
```

## Update Porting/config.sh and Porting/config\_H

[XXX This section needs revision. We're currently working on easing the task of keeping the vms, win32, and plan9 config.sh info up-to-date. The plan is to use keep up-to-date 'canned' config.sh files in the appropriate subdirectories and then generate 'canned' config.h files for vms, win32, etc. from the generic config.sh file. This is to ease maintenance. When Configure gets updated, the parts sometimes get scrambled around, and the changes in config\_H can sometimes be very hard to follow. config.sh, on the other hand, can safely be sorted, so it's easy to track (typically very small) changes to config.sh and then propagate them to a canned 'config.h' by any number of means, including a perl script in win32/ or carrying config.sh and config\_h.SH to a Unix system and running sh config\_h.SH.) Vms uses configure.com to generate its own config.sh and config.h. If you want to add a new variable to config.sh check with vms folk how to add it to configure.com too. XXX]

The Porting/config.sh and Porting/config\_H files are provided to help those folks who can't run Configure. It is important to keep them up-to-date. If you have changed config\_h.SH, those changes must be reflected in config\_H as well. (The name config\_H was chosen to distinguish the file from config.h even on case-insensitive file systems.) Simply edit the existing config\_H file; keep the first few explanatory lines and then copy your new config.h below.

It may also be necessary to update win32/config.?c, and plan9/config.plan9, though you should be quite careful in doing so if you are not familiar with those systems. You might want to issue your patch with a promise to quickly issue a follow-up that handles those directories.

## make run\_byacc

If you have byacc-1.8.2 (available from CPAN), and if there have been changes to *perly.y*, you can regenerate the *perly.c* file. The run\_byacc makefile target does this by running byacc and then applying some patches so that byacc dynamically allocates space, rather than having fixed limits. This patch is handled by the *perly.fixer* script. Depending on the nature of the changes to *perly.y*, you may or may not have to hand-edit the patch to apply correctly. If you do, you should include the edited patch in the new distribution. If you have byacc-1.9, the patch won't apply cleanly. Changes to the printf output statements

mean the patch won't apply cleanly. Long ago I started to fix *perly\_fixer* to detect this, but I never completed the task.

If *perly.c* or *perly.h* changes, make sure you run `perl vms/vms_yfix.pl` to update the corresponding VMS files. This could be taken care of by the `regen_all` target in the Unix Makefile. See also [VMS-specific updates](#).

Some additional notes from Larry on this:

Don't forget to regenerate *perly\_c.diff*.

```
byacc -d perly.y
mv y.tab.c perly.c
patch perly.c <perly_c.diff
manually apply any failed hunks
diff -c perly.c.orig perly.c >perly_c.diff
```

One chunk of lines that often fails begins with

```
#line 29 "perly.y"
```

and ends one line before

```
#define YYERRCODE 256
```

This only happens when you add or remove a token type. I suppose this could be automated, but it doesn't happen very often nowadays.

Larry

### make regen\_all

This target takes care of the `PERLYVMS`, `regen_headers`, and `regen_pods` targets.

### make regen\_headers

The *embed.h*, *keywords.h*, and *opcode.h* files are all automatically generated by perl scripts. Since the user isn't guaranteed to have a working perl, we can't require the user to generate them. Hence you have to, if you're making a distribution.

I used to include rules like the following in the makefile:

```
The following three header files are generated automatically
The correct versions should be already supplied with the perl kit,
in case you don't have perl or 'sh' available.
The - is to ignore error return codes in case you have the source
installed read-only or you don't have perl yet.
keywords.h: keywords.pl
 @echo "Don't worry if this fails."
 - perl keywords.pl
```

However, I got **lots** of mail consisting of people worrying because the command failed. I eventually decided that I would save myself time and effort by manually running `make regen_headers` myself rather than answering all the questions and complaints about the failing command.

### make regen\_pods

Will run 'make regen\_pods' in the pod directory for indexing.

### global.sym, interp.sym and perlio.sym

Make sure these files are up-to-date. Read the comments in these files and in *perl\_exp.SH* to see what to do.

## Binary compatibility

If you do change *global.sym* or *interp.sym*, think carefully about what you are doing. To the extent reasonable, we'd like to maintain source and binary compatibility with older releases of perl. That way, extensions built under one version of perl will continue to work with new versions of perl.

Of course, some incompatible changes may well be necessary. I'm just suggesting that we not make any such changes without thinking carefully about them first. If possible, we should provide backwards-compatibility stubs. There's a lot of XS code out there. Let's not force people to keep changing it.

## Changes

Be sure to update the *Changes* file. Try to include both an overall summary as well as detailed descriptions of the changes. Your audience will include other developers and users, so describe user-visible changes (if any) in terms they will understand, not in code like "initialize foo variable in bar function".

There are differing opinions on whether the detailed descriptions ought to go in the Changes file or whether they ought to be available separately in the patch file (or both). There is no disagreement that detailed descriptions ought to be easily available somewhere.

If you update the subversion number in *patchlevel.h*, you may need to change the version number near the top of the *Changes* file.

## Todo

The *Todo* file contains a roughly-categorized unordered list of aspects of Perl that could use enhancement, features that could be added, areas that could be cleaned up, and so on. During your term as pumpkin-holder, you will probably address some of these issues, and perhaps identify others which, while you decide not to address them this time around, may be tackled in the future. Update the file reflect the situation as it stands when you hand over the pumpkin.

You might like, early in your pumpkin-holding career, to see if you can find champions for particular issues on the to-do list: an issue owned is an issue more likely to be resolved.

There are also some more porting-specific *Todo* items later in this file.

## OS/2-specific updates

In the os2 directory is *diff.configure*, a set of OS/2-specific diffs against *Configure*. If you make changes to *Configure*, you may want to consider regenerating this diff file to save trouble for the OS/2 maintainer.

You can also consider the OS/2 diffs as reminders of portability things that need to be fixed in *Configure*.

## VMS-specific updates

If you have changed *perly.y* or *perly.c*, then you most probably want to update *vms/perly\_{h,c}.vms* by running `perl vms/vms_yfix.pl`, or by running 'make regen\_all' which will run that script for you.

The Perl revision number appears as "perl5" in *configure.com*. It is courteous to update that if necessary.

## Making the new distribution

Suppose, for example, that you want to make version 5.004\_08. Then you can do something like the following

```
mkdir ../perl5.004_08
awk '{print $1}' MANIFEST | cpio -pdm ../perl5.004_08
cd ../
tar cf perl5.004_08.tar perl5.004_08
gzip --best perl5.004_08.tar
```

These steps, with extra checks, are automated by the *Porting/makerel* script.



## Making a new patch

I find the *makepatch* utility quite handy for making patches. You can obtain it from any CPAN archive under [http://www.cpan.org/authors/Johan\\_Vromans/](http://www.cpan.org/authors/Johan_Vromans/). There are a couple of differences between my version and the standard one. I have mine do a

```
Print a reassuring "End of Patch" note so people won't
wonder if their mailer truncated patches.
print "\n\nEnd of Patch.\n";
```

at the end. That's because I used to get questions from people asking if their mail was truncated.

It also writes Index: lines which include the new directory prefix (change Index: print, approx line 294 or 310 depending on the version, to read: `print PATCH ("Index: $newdir$new\n");`). That helps patches work with more POSIX conformant patch programs.

Here's how I generate a new patch. I'll use the hypothetical 5.004\_07 to 5.004\_08 patch as an example.

```
unpack perl5.004_07/
gzip -d -c perl5.004_07.tar.gz | tar -xof -
unpack perl5.004_08/
gzip -d -c perl5.004_08.tar.gz | tar -xof -
makepatch perl5.004_07 perl5.004_08 > perl5.004_08.pat
```

Makepatch will automatically generate appropriate **rm** commands to remove deleted files. Unfortunately, it will not correctly set permissions for newly created files, so you may have to do so manually. For example, patch 5.003\_04 created a new test *t/op/gv.t* which needs to be executable, so at the top of the patch, I inserted the following lines:

```
Make a new test
touch t/op/gv.t
chmod +x t/opt/gv.t
```

Now, of course, my patch is now wrong because makepatch didn't know I was going to do that command, and it patched against /dev/null.

So, what I do is sort out all such shell commands that need to be in the patch (including possible mv-ing of files, if needed) and put that in the shell commands at the top of the patch. Next, I delete all the patch parts of perl5.004\_08.pat, leaving just the shell commands. Then, I do the following:

```
cd perl5.004_07
sh ../perl5.004_08.pat
cd ..
makepatch perl5.004_07 perl5.004_08 >> perl5.004_08.pat
```

(Note the append to preserve my shell commands.) Now, my patch will line up with what the end users are going to do.

## Testing your patch

It seems obvious, but be sure to test your patch. That is, verify that it produces exactly the same thing as your full distribution.

```
rm -rf perl5.004_07
gzip -d -c perl5.004_07.tar.gz | tar -xf -
cd perl5.004_07
sh ../perl5.004_08.pat
patch -p1 -N < ../perl5.004_08.pat
cd ..
gdiff -r perl5.004_07 perl5.004_08
```

where **gdiff** is GNU diff. Other diff's may also do recursive checking.

## More testing

Again, it's obvious, but you should test your new version as widely as you can. You can be sure you'll hear about it quickly if your version doesn't work on both ANSI and pre-ANSI compilers, and on common systems such as SunOS 4.1.[34], Solaris, and Linux.

If your changes include conditional code, try to test the different branches as thoroughly as you can. For example, if your system supports dynamic loading, you can also test static loading with

```
sh Configure -Uusedl
```

You can also hand-tweak your config.h to try out different #ifdef branches.

## Other tests

### CHECK\_FORMAT

To test the correct use of printf-style arguments, Configure with `-Dccflags='-DCHECK_FORMAT -Wformat'` and run `make`. The compiler will produce warning of incorrect use of format arguments. `CHECK_FORMAT` changes perl-defined formats to common formats, so DO NOT USE the executable produced by this process.

A more accurate approach is the following commands:

- build miniperl with `-DCHECK_FORMAT`

```
make clean
make miniperl OPTIMIZE=-DCHECK_FORMAT >& mini.log
```
- build a clean miniperl, and build everything else from that with `-DCHECK_FORMAT`

```
make clean
make miniperl
make all OPTIMIZE='-DCHECK_FORMAT -Wformat' >& make.log
```
- clean up, and print warnings from the log files

```
make clean
perl -nwe 'print if /^S+:/ and not /^make\b/' \
mini.log make.log
```

(`-Wformat` support by Robin Barker.)

## Running Purify

Purify is a commercial tool that is helpful in identifying memory overruns, wild pointers, memory leaks and other such badness. Perl must be compiled in a specific way for optimal testing with Purify.

Use the following commands to test perl with Purify:

```
sh Configure -des -Doptimize=-g -Uusemymalloc -Dusemultiplicity \
-Dccflags=-DPURIFY
setenv PURIFYOPTIONS "-chain-length=25"
make all pureperl
cd t
ln -s ../pureperl perl
setenv PERL_DESTRUCT_LEVEL 2
./perl TEST
```

Disabling Perl's malloc allows Purify to monitor allocations and leaks more closely; using Perl's malloc will make Purify report most leaks in the "potential" leaks category. Enabling the multiplicity option allows perl to clean up thoroughly when the interpreter shuts down, which reduces the number of bogus leak reports from Purify. The `-DPURIFY` enables any Purify-specific debugging code in the sources.

Purify outputs messages in "Viewer" windows by default. If you don't have a windowing environment or if you simply want the Purify output to unobtrusively go to a log file instead of to the interactive window, use the following options instead:

```
setenv PURIFYOPTIONS "-chain-length=25 -windows=no -log-file=perl.log \
-append-logfile=yes"
```

The only currently known leaks happen when there are compile-time errors within eval or require. (Fixing these is non-trivial, unfortunately, but they must be fixed eventually.)

### Common Gotcha's

**#elif** The `#elif` preprocessor directive is not understood on all systems. Specifically, I know that Pyramids don't understand it. Thus instead of the simple

```
#if defined(I_FOO)
include <foo.h>
#elif defined(I_BAR)
include <bar.h>
#else
include <fubar.h>
#endif
```

You have to do the more Byzantine

```
#if defined(I_FOO)
include <foo.h>
#else
if defined(I_BAR)
include <bar.h>
else
include <fubar.h>
endif
#endif
```

Incidentally, whitespace between the leading `#` and the preprocessor command is not guaranteed, but is very portable and you may use it freely. I think it makes things a bit more readable, especially once things get rather deeply nested. I also think that things should almost never get too deeply nested, so it ought to be a moot point :-)

### Probably Prefer POSIX

It's often the case that you'll need to choose whether to do something the BSD-ish way or the POSIX-ish way. It's usually not a big problem when the two systems use different names for similar functions, such as `memcmp()` and `bcmp()`. The `perl.h` header file handles these by appropriate `#defines`, selecting the POSIX `mem*` functions if available, but falling back on the `b*` functions, if need be.

More serious is the case where some brilliant person decided to use the same function name but give it a different meaning or calling sequence :-). `getpgrp()` and `setpgrp()` come to mind. These are a real problem on systems that aim for conformance to one standard (e.g. POSIX), but still try to support the other way of doing things (e.g. BSD). My general advice (still not really implemented in the source) is to do something like the following. Suppose there are two alternative versions, `fooPOSIX()` and `fooBSD()`.

```
#ifdef HAS_FOOPOSIX
/* use fooPOSIX(); */
#else
ifdef HAS_FOOBSD
/* try to emulate fooPOSIX() with fooBSD();
perhaps with the following: */
```

```
define fooPOSIX fooBSD
else
/* Uh, oh. We have to supply our own. */
define fooPOSIX Perl_fooPOSIX
endif
#endif
```

### Think positively

If you need to add an `#ifdef` test, it is usually easier to follow if you think positively, e.g.

```
#ifdef HAS_NEATO_FEATURE
 /* use neato feature */
#else
 /* use some fallback mechanism */
#endif
```

rather than the more impenetrable

```
#ifndef MISSING_NEATO_FEATURE
 /* Not missing it, so we must have it, so use it */
#else
 /* Are missing it, so fall back on something else. */
#endif
```

Of course for this toy example, there's not much difference. But when the `#ifdef`'s start spanning a couple of screen fulls, and the `#else`'s are marked something like

```
#else /* !MISSING_NEATO_FEATURE */
```

I find it easy to get lost.

### Providing Missing Functions — Problem

Not all systems have all the neat functions you might want or need, so you might decide to be helpful and provide an emulation. This is sound in theory and very kind of you, but please be careful about what you name the function. Let me use the `pause()` function as an illustration.

Perl5.003 has the following in *perl.h*

```
#ifndef HAS_PAUSE
#define pause() sleep((32767<<16)+32767)
#endif
```

Configure sets `HAS_PAUSE` if the system has the `pause()` function, so this `#define` only kicks in if the `pause()` function is missing. Nice idea, right?

Unfortunately, some systems apparently have a prototype for `pause()` in *unistd.h*, but don't actually have the function in the library. (Or maybe they do have it in a library we're not using.)

Thus, the compiler sees something like

```
extern int pause(void);
/* . . . */
#define pause() sleep((32767<<16)+32767)
```

and dies with an error message. (Some compilers don't mind this; others apparently do.)

To work around this, 5.003\_03 and later have the following in *perl.h*:

```
/* Some unistd.h's give a prototype for pause() even though
 HAS_PAUSE ends up undefined. This causes the #define
 below to be rejected by the compiler. Sigh.
*/
#ifdef HAS_PAUSE
```

```
define Pause
#else
define Pause() sleep((32767<<16)+32767)
#endif
```

This works.

The curious reader may wonder why I didn't do the following in *util.c* instead:

```
#ifndef HAS_PAUSE
void pause()
{
 sleep((32767<<16)+32767);
}
#endif
```

That is, since the function is missing, just provide it. Then things would probably be been alright, it would seem.

Well, almost. It could be made to work. The problem arises from the conflicting needs of dynamic loading and namespace protection.

For dynamic loading to work on AIX (and VMS) we need to provide a list of symbols to be exported. This is done by the script *perl\_exp.SH*, which reads *global.sym* and *interp.sym*. Thus, the pause symbol would have to be added to *global.sym*. So far, so good.

On the other hand, one of the goals of Perl5 is to make it easy to either extend or embed perl and link it with other libraries. This means we have to be careful to keep the visible namespace "clean". That is, we don't want perl's global variables to conflict with those in the other application library. Although this work is still in progress, the way it is currently done is via the *embed.h* file. This file is built from the *global.sym* and *interp.sym* files, since those files already list the globally visible symbols. If we had added pause to *global.sym*, then *embed.h* would contain the line

```
#define pause Perl_pause
```

and calls to pause in the perl sources would now point to Perl\_pause. Now, when *ld* is run to build the *perl* executable, it will go looking for perl\_pause, which probably won't exist in any of the standard libraries. Thus the build of perl will fail.

Those systems where HAS\_PAUSE is not defined would be ok, however, since they would get a Perl\_pause function in util.c. The rest of the world would be in trouble.

And yes, this scenario has happened. On SCO, the function chsize is available. (I think it's in *-lx*, the Xenix compatibility library.) Since the perl4 days (and possibly before), Perl has included a chsize function that gets called something akin to

```
#ifndef HAS_CHSIZE
I32 chsize(fd, length)
/* . . . */
#endif
```

When 5.003 added

```
#define chsize Perl_chsize
```

to *embed.h*, the compile started failing on SCO systems.

The "fix" is to give the function a different name. The one implemented in 5.003\_05 isn't optimal, but here's what was done:

```
#ifdef HAS_CHSIZE
ifdef my_chsize /* Probably #defined to Perl_my_chsize in embed.h */
undef my_chsize
```

```
endif
define my_chsize chsize
#endif
```

My explanatory comment in patch 5.003\_05 said:

```
Undef and then re-define my_chsize from Perl_my_chsize to
just plain chsize if this system HAS_CHSIZE. This probably only
applies to SCO. This shows the perils of having internal
functions with the same name as external library functions :-).
```

Now, we can safely put my\_chsize in *global.sym*, export it, and hide it with *embed.h*.

To be consistent with what I did for pause, I probably should have called the new function Chsize, rather than my\_chsize. However, the perl sources are quite inconsistent on this (Consider New, Mymalloc, and Myrealloc, to name just a few.)

There is a problem with this fix, however, in that Perl\_chsize was available as a *libperl.a* library function in 5.003, but it isn't available any more (as of 5.003\_07). This means that we've broken binary compatibility. This is not good.

#### Providing missing functions — some ideas

We currently don't have a standard way of handling such missing function names. Right now, I'm effectively thinking aloud about a solution. Some day, I'll try to formally propose a solution.

Part of the problem is that we want to have some functions listed as exported but not have their names mangled by embed.h or possibly conflict with names in standard system headers. We actually already have such a list at the end of *perl\_exp.SH* (though that list is out-of-date):

```
extra globals not included above.
cat <<END >> perl.exp
perl_init_ext
perl_init_fold
perl_init_i18nl14n
perl_alloc
perl_construct
perl_destruct
perl_free
perl_parse
perl_run
perl_get_sv
perl_get_av
perl_get_hv
perl_get_cv
perl_call_argv
perl_call_pv
perl_call_method
perl_call_sv
perl_requirepv
safecalloc
safemalloc
saferealloc
safefree
```

This still needs much thought, but I'm inclined to think that one possible solution is to prefix all such functions with perl\_ in the source and list them along with the other perl\_\* functions in *perl\_exp.SH*.

Thus, for chsize, we'd do something like the following:

```
/* in perl.h */
#ifdef HAS_CHSIZE
define perl_chsize chsize
#endif
```

then in some file (e.g. *util.c* or *doio.c*) do

```
#ifndef HAS_CHSIZE
I32 perl_chsize(fd, length)
/* implement the function here . . . */
#endif
```

Alternatively, we could just always use *chsize* everywhere and move *chsize* from *global.sym* to the end of *perl\_exp.SH*. That would probably be fine as long as our *chsize* function agreed with all the *chsize* function prototypes in the various systems we'll be using. As long as the prototypes in actual use don't vary that much, this is probably a good alternative. (As a counter-example, note how *Configure* and *perl* have to go through hoops to find and use *get Malloc\_t* and *Free\_t* for *malloc* and *free*.)

At the moment, this latter option is what I tend to prefer.

### All the world's a VAX

Sorry, showing my age:–). Still, all the world is not BSD 4.[34], SVR4, or POSIX. Be aware that SVR3–derived systems are still quite common (do you have any idea how many systems run SCO?) If you don't have a bunch of v7 manuals handy, the metaconfig units (by default installed in */usr/local/lib/dist/U*) are a good resource to look at for portability.

## Miscellaneous Topics

### Autoconf

Why does *perl* use a metaconfig–generated *Configure* script instead of an autoconf–generated *configure* script?

Metaconfig and autoconf are two tools with very similar purposes. Metaconfig is actually the older of the two, and was originally written by Larry Wall, while autoconf is probably now used in a wider variety of packages. The autoconf info file discusses the history of autoconf and how it came to be. The curious reader is referred there for further information.

Overall, both tools are quite good, I think, and the choice of which one to use could be argued either way. In March, 1994, when I was just starting to work on *Configure* support for Perl5, I considered both autoconf and metaconfig, and eventually decided to use metaconfig for the following reasons:

#### Compatibility with Perl4

Perl4 used metaconfig, so many of the *#ifdef*'s were already set up for metaconfig. Of course metaconfig had evolved some since Perl4's days, but not so much that it posed any serious problems.

#### Metaconfig worked for me

My system at the time was Interactive 2.2, an SVR3.2/386 derivative that also had some POSIX support. Metaconfig–generated *Configure* scripts worked fine for me on that system. On the other hand, autoconf–generated scripts usually didn't. (They did come quite close, though, in some cases.) At the time, I actually fetched a large number of GNU packages and checked. Not a single one configured and compiled correctly out–of–the–box with the system's *cc* compiler.

#### Configure can be interactive

With both autoconf and metaconfig, if the script works, everything is fine. However, one of my main problems with autoconf–generated scripts was that if it guessed wrong about something, it could be **very** hard to go back and fix it. For example, autoconf always insisted on passing the *-Xp* flag to *cc* (to turn on POSIX behavior), even when that wasn't what I wanted or needed for that package. There was no way short of editing the *configure* script to turn this off. You couldn't just edit the resulting *Makefile* at the end because the *-Xp* flag influenced a number of other *configure* tests.

Metaconfig's Configure scripts, on the other hand, can be interactive. Thus if Configure is guessing things incorrectly, you can go back and fix them. This isn't as important now as it was when we were actively developing Configure support for new features such as dynamic loading, but it's still useful occasionally.

### GPL

At the time, autoconf-generated scripts were covered under the GNU Public License, and hence weren't suitable for inclusion with Perl, which has a different licensing policy. (Autoconf's licensing has since changed.)

### Modularity

Metaconfig builds up Configure from a collection of discrete pieces called "units". You can override the standard behavior by supplying your own unit. With autoconf, you have to patch the standard files instead. I find the metaconfig "unit" method easier to work with. Others may find metaconfig's units clumsy to work with.

### Why isn't there a directory to override Perl's library?

Mainly because no one's gotten around to making one. Note that "making one" involves changing perl.c, Configure, config\_h.SH (and associated files, see above), and *documenting* it all in the INSTALL file.

Apparently, most folks who want to override one of the standard library files simply do it by overwriting the standard library files.

### APPLLIB

In the perl.c sources, you'll find an undocumented APPLLIB\_EXP variable, sort of like PRIVLIB\_EXP and ARCHLIB\_EXP (which are documented in config\_h.SH). Here's what APPLLIB\_EXP is for, from a mail message from Larry:

The main intent of APPLLIB\_EXP is for folks who want to send out a version of Perl embedded in their product. They would set the symbol to be the name of the library containing the files needed to run or to support their particular application. This works at the "override" level to make sure they get their own versions of any library code that they absolutely must have configuration control over.

As such, I don't see any conflict with a sysadmin using it for a override-ish sort of thing, when installing a generic Perl. It should probably have been named something to do with overriding though. Since it's undocumented we could still change it... :-)

Given that it's already there, you can use it to override distribution modules. If you do

```
sh Configure -Dccflags='-DAPPLLIB_EXP=/my/override'
```

then perl.c will put /my/override ahead of ARCHLIB and PRIVLIB.

### Shared libperl.so location

Why isn't the shared libperl.so installed in /usr/lib/ along with "all the other" shared libraries? Instead, it is installed in \$archlib, which is typically something like

```
/usr/local/lib/perl5/archname/5.00404
```

and is architecture- and version-specific.

The basic reason why a shared libperl.so gets put in \$archlib is so that you can have more than one version of perl on the system at the same time, and have each refer to its own libperl.so.

Three examples might help. All of these work now; none would work if you put libperl.so in /usr/lib.



1. Suppose you want to have both threaded and non-threaded perl versions around. Configure will name both perl libraries "libperl.so" (so that you can link to them with `-lperl`). The perl binaries tell them apart by having looking in the appropriate `$archlib` directories.
2. Suppose you have perl5.004\_04 installed and you want to try to compile it again, perhaps with different options or after applying a patch. If you already have libperl.so installed in `/usr/lib/`, then it may be either difficult or impossible to get `ld.so` to find the new libperl.so that you're trying to build. If, instead, libperl.so is tucked away in `$archlib`, then you can always just change `$archlib` in the current perl you're trying to build so that `ld.so` won't find your old libperl.so. (The INSTALL file suggests you do this when building a debugging perl.)
3. The shared perl library is not a "well-behaved" shared library with proper major and minor version numbers, so you can't necessarily have perl5.004\_04 and perl5.004\_05 installed simultaneously. Suppose perl5.004\_04 were to install `/usr/lib/libperl.so.4.4`, and perl5.004\_05 were to install `/usr/lib/libperl.so.4.5`. Now, when you try to run perl5.004\_04, `ld.so` might try to load libperl.so.4.5, since it has the right "major version" number. If this works at all, it almost certainly defeats the reason for keeping perl5.004\_04 around. Worse, with development subversions, you certainly can't guarantee that libperl.so.4.4 and libperl.so.4.55 will be compatible.

Anyway, all this leads to quite obscure failures that are sure to drive casual users crazy. Even experienced users will get confused :-). Upon reflection, I'd say leave libperl.so in `$archlib`.

4. Indentation style: over the years Perl has become a mishmash of various indentation styles, but the original "Larry style" can probably be restored with (GNU) indent somewhat like this:

```
indent -kr -nce -psl -sc
```

A more ambitious solution would also specify a list of Perl specific types with `-TSV -TAV -THV .. -TMAGIC -TPerIO ...` but that list would be quite ungainly. Also note that GNU indent also doesn't do aligning of consecutive assignments, which would truly wreck the layout in places like `sv.c:Perl_sv_upgrade()` or `sv.c:Perl_clone_using()`. Similarly nicely aligned `&&s`, `||s` and `==s` would not be respected.

### Upload Your Work to CPAN

You can upload your work to CPAN if you have a CPAN id. Check out <http://www.cpan.org/modules/04pause.html> for information on `_PAUSE_`, the Perl Author's Upload Server.

I typically upload both the patch file, e.g. *perl5.004\_08.pat.gz* and the full tar file, e.g. *perl5.004\_08.tar.gz*.

If you want your patch to appear in the *src/5.0/unsupported* directory on CPAN, send e-mail to the CPAN master librarian. (Check out <http://www.cpan.org/CPAN.html> ).

### Help Save the World

You should definitely announce your patch on the perl5-porters list. You should also consider announcing your patch on `comp.lang.perl.announce`, though you should make it quite clear that a subversion is not a production release, and be prepared to deal with people who will not read your disclaimer.

### Todo

Here, in no particular order, are some Configure and build-related items that merit consideration. This list isn't exhaustive, it's just what I came up with off the top of my head.

### Adding missing library functions to Perl

The perl Configure script automatically determines which headers and functions you have available on your system and arranges for them to be included in the compilation and linking process. Occasionally, when porting perl to an operating system for the first time, you may find that the operating system is missing a key function. While perl may still build without this function, no perl program will be able to reference the missing function. You may be able to write the missing function yourself, or you may be able to find the missing function in the distribution files for another software package. In this case, you need to instruct the perl configure-and-build process to use your function. Perform these steps.

- Code and test the function you wish to add. Test it carefully; you will have a much easier time debugging your code independently than when it is a part of perl.
- Here is an implementation of the POSIX truncate function for an operating system (VOS) that does not supply one, but which does supply the `ftruncate()` function.

```
/* Beginning of modification history */
/* Written 02-01-02 by Nick Ing-Simmons (nick@ing-simmons.net) */
/* End of modification history */

/* VOS doesn't supply a truncate function, so we build one up
 from the available POSIX functions. */

#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

int
truncate(const char *path, off_t len)
{
 int fd = open(path, O_WRONLY);
 int code = -1;
 if (fd >= 0) {
 code = ftruncate(fd, len);
 close(fd);
 }
 return code;
}
```

Place this file into a subdirectory that has the same name as the operating system. This file is named `perl/vos/vos.c`

- If your operating system has a hints file (in `perl/hints/XXX.sh` for an operating system named XXX), then start with it. If your operating system has no hints file, then create one. You can use a hints file for a similar operating system, if one exists, as a template.
- Add lines like the following to your hints file. The first line (`d_truncate="define"`) instructs Configure that the `truncate()` function exists. The second line (`archobjs="vos.o"`) instructs the makefiles that the perl executable depends on the existence of a file named "vos.o". (Make will automatically look for "vos.c" and compile it with the same options as the perl source code). The final line ("`test -h...`") adds a symbolic link to the top-level directory so that make can find `vos.c`. Of course, you should use your own operating system name for the source file of extensions, not "vos.c".

```
VOS does not havetruncate() but we supply one in vos.c
d_truncate="define"
archobjs="vos.o"

Help gmake find vos.c
test -h vos.c || ln -s vos/vos.c vos.c
```

The hints file is a series of shell commands that are run in the top-level directory (the "perl" directory). Thus, these commands are simply executed by Configure at an appropriate place during its execution.

- At this point, you can run the Configure script and rebuild perl. Carefully test the newly-built perl to ensure that normal paths, and error paths, behave as you expect.

### Good ideas waiting for round tuits

Configure `-Dsrc=/blah/blah`

We should be able to emulate **configure** `—sreaddir`. Tom Tromey [tromey@creche.cygnum.com](mailto:tromey@creche.cygnum.com) has submitted some patches to the dist-users mailing list along these lines. They have been folded back

into the main distribution, but various parts of the perl Configure/build/install process still assume `src='.'`.

#### Hint file fixes

Various hint files work around Configure problems. We ought to fix Configure so that most of them aren't needed.

#### Hint file information

Some of the hint file information (particularly dynamic loading stuff) ought to be fed back into the main metaconfig distribution.

### Probably good ideas waiting for round tuits

#### GNU configure —options

I've received sensible suggestions for `--exec_prefix` and other GNU configure —options. It's not always obvious exactly what is intended, but this merits investigation.

#### make clean

Currently, **make clean** isn't all that useful, though **make realclean** and **make distclean** are. This needs a bit of thought and documentation before it gets cleaned up.

#### Try gcc if cc fails

Currently, we just give up.

#### bypassing safe\*alloc wrappers

On some systems, it may be safe to call the system malloc directly without going through the util.c safe\* layers. (Such systems would accept free(0), for example.) This might be a time-saver for systems that already have a good malloc. (Recent Linux libc's apparently have a nice malloc that is well-tuned for the system.)

### Vague possibilities

#### MacPerl

Get some of the Macintosh stuff folded back into the main distribution.

#### gconvert replacement

Maybe include a replacement function that doesn't lose data in rare cases of coercion between string and numerical values.

#### Improve makedepend

The current makedepend process is clunky and annoyingly slow, but it works for most folks. Alas, it assumes that there is a filename `$firstmakefile` that the **make** command will try to use before it uses **Makefile**. Such may not be the case for all **make** commands, particularly those on non-Unix systems.

Probably some variant of the BSD **.depend** file will be useful. We ought to check how other packages do this, if they do it at all. We could probably pre-generate the dependencies (with the exception of malloc.o, which could probably be determined at **Makefile.SH** extraction time).

#### GNU Makefile standard targets

GNU software generally has standardized Makefile targets. Unless we have good reason to do otherwise, I see no reason not to support them.

#### File locking

Somehow, straighten out, document, and implement `lockf()`, `flock()`, and/or `fcntl()` file locking. It's a mess. See `$d_fcntl_can_lock` in recent config.sh files though.

**AUTHORS**

Original author: Andy Dougherty [doughera@lafcol.lafayette.edu](mailto:doughera@lafcol.lafayette.edu) . Additions by Chip Salzenberg [chip@perl.com](mailto:chip@perl.com) and Tim Bunce [Tim.Bunce@ig.co.uk](mailto:Tim.Bunce@ig.co.uk) .

All opinions expressed herein are those of the author(s).

**LAST MODIFIED**

\$Id: pumpkin.pod,v 1.23 2000/01/13 19:45:13 doughera Released \$

## NAME

repository – Using the Perl repository

## Synopsis

First, we assume here that you have already decided that you will need **write** access to the repository. If all you need is **read** access, there are much better ways to access the most current state of the perl repository, or explore individual files and patches therein. See [perlhack](#) for details.

This document describes what a Perl Porter needs to do to start using the Perl repository.

## Prerequisites

You'll need to get hold of the following software.

### Perforce

Download a perforce client from:

<http://www.perforce.com/perforce/loadprog.html>

You'll probably also want to look at:

<http://www.perforce.com/perforce/technical.html>

where you can look at or download its documentation.

**ssh** If you don't already have access to an ssh client, then look at its home site

<http://www.cs.hut.fi/ssh> which mentions ftp sites from which it's available. You only need to build the client parts (ssh and ssh-keygen should suffice).

## Creating an SSH Key Pair

If you already use ssh and want to use the same key pair for perl repository access then you can skip the rest of this section. Otherwise, generate an ssh key pair for use with the repository by typing the command

```
ssh-keygen
```

After generating a key pair and testing it, ssh-keygen will ask you to enter a filename in which to save the key. The default it offers will be the file `~/.ssh/identity` which is suitable unless you particularly want to keep separate ssh identities for some reason. If so, you could save the perl repository private key in the file `~/.ssh/perl`, for example, but I will use the standard filename in the remainder of the examples of this document.

After typing in the filename, it will prompt you to type in a passphrase. The private key will itself be encrypted so that it is usable only when that passphrase is typed. (When using ssh, you will be prompted when it requires a pass phrase to unlock a private key.) If you provide a blank passphrase then no passphrase will be needed to unlock the key and, as a consequence, anyone who gains access to the key file gains access to accounts protected with that key (barring additional configuration to restrict access by IP address).

When you have typed the passphrase in twice, ssh-keygen will confirm where it has saved the private key (in the filename you gave and with permissions set to be only readable by you), what your public key is (don't worry: you don't need to memorise it) and where it has saved the corresponding public key. The public key is saved in a filename corresponding to your private key's filename but with ".pub" appended, usually `~/.ssh/identity.pub`. That public key can be (but need not be) world readable. It is not used by your own system at all.

## Notifying the Repository Keeper

Mail the contents of that public key file to the keeper of the perl repository (see [/Contact Information](#) below). When the key is added to the repository host's configuration file, you will be able to connect to it with ssh by using the corresponding private key file (after unlocking it with your chosen passphrase).

## Connecting to the Repository

Connections to the repository are made by using ssh to provide a TCP "tunnel" rather than by using ssh to login to or invoke any ordinary commands on the repository. When you want to start a session using the repository, use the command

```
ssh -l perlrep -f -q -x -L 1666:127.0.0.1:1666 sickle.activestate.com foo
```

If you are not using the default filename of `~/.ssh/identity` to hold your perl repository private key then you'll need to add the option `-i filename` to tell ssh where it is. Unless you chose a blank passphrase for that private key, ssh will prompt you for the passphrase to unlock that key. Then ssh will fork and put itself in the background, returning you (silently) to your shell prompt. The tunnel for repository access is now ready for use.

For the sake of completeness (and for the case where the chosen port of 1666 is already in use on your machine), I'll briefly describe what all those ssh arguments are for.

### **-l perlrep**

Use a remote username of perlrep. (The account on the repository which provides the end-point of the ssh tunnel is named "perlrep".)

**-f** Tells ssh to fork and remain running in the background. Since ssh is only being used for its tunnelling capabilities, the command that ssh runs never does any I/O and can sit silently in the background.

**-q** Tells ssh to be quiet. Without this option, ssh will output a message each time you use a p4 command (since each p4 command tunnels over the ssh connection to reach the repository).

**-x** Tells ssh not to bother to set up a tunnel for X11 connections. The repository doesn't allow this anyway.

### **-L 1666:127.0.0.1:1666**

This is the important option. It tells ssh to listen out for connections made to port 1666 on your local machine. When such a connection is made, the ssh client tells the remote side (the corresponding ssh daemon on the repository) to make a connection to IP address 127.0.0.1, port 1666. Data flowing along that connection is tunnelled over the ssh connection (encrypted). The perforce daemon running on the repository only accepts connections from localhost and that is exactly where ssh-tunnelled connections appear to come from.

If port 1666 is already in use on your machine then you can choose any non-privileged port (a number between 1024 and 65535) which happens to be free on your machine. It's the first of the three colon separated values that you should change. Picking port 2345 would mean changing the option to **-L 2345:127.0.0.1:1666**. Whatever port number you choose should be used for the value of the P4PORT environment variable (q.v.).

sickle.activestate.com

This is the canonical name of the host on which the perl repository resides. Its IP address is 199.60.48.20.

foo This is a dummy place holder argument. Without an argument here, ssh will try to perform an interactive login to the repository which is not allowed. Ordinarily, this argument is for the one-off command which is to be executed on the remote host. However, the repository's ssh configuration file uses the "command=" option to force a particular command to run so the actual value of the argument is ignored. The command that's actually run merely pauses and waits for the ssh connection to drop, then exits.

## Problems

You should normally get a prompt that asks for the passphrase for your RSA key when you connect with the ssh command shown above. If you see a prompt that looks like:

```
perlrep@sickle.activestate.com's password:
```

Then you either don't have a `~/.ssh/identity` file corresponding to your public key, or your `~/.ssh/identity` file is not readable. Fix the problem and try again.

## Using the Perforce Client

Remember to read the documentation for Perforce. You need to make sure that three environment variable are set correctly before using the `p4` client with the perl repository.

### P4PORT

Set this to `localhost:1666` (the port for your ssh client to listen on) unless that port is already in use on your host. If it is, see the section above on the `-L 1666:127.0.0.1:1666` option to `ssh`.

### P4CLIENT

The value of this is the name by which Perforce knows your host's workspace. You need to pick a name (normally, your Perforce username, a dash, and your hostname) when you first start using the perl repository and then stick with it.

Perforce keeps track of the files you have on your machine. It does this through your client. When you first sync a version of a file, the file comes from the server to your machine. If you sync the same file again the server does nothing because it knows you already have the file.

You should NOT use the same client on different machines. If you do you probably won't get the files you expect, and may end up with nasty corruption. Perforce allows you to have as many clients as you want. For example, `sally-home`, `sally-openbsd`, `sally-laptop`.

Also, never change the client's root and view at the same time. See

[http://www.perforce.com/perforce/doc.002/manuals/p4guide/04\\_details.html#1048341](http://www.perforce.com/perforce/doc.002/manuals/p4guide/04_details.html#1048341)

If you have multiple hosts sharing the same directory structure via NFS then you may be able to get away with only one client name, but be careful.

The `p4 clients` command lists all currently known clients.

### P4USER

This is the username by which perforce knows you. Use your username if you have a well known or obvious one or else pick a new one which other `perl5-porters` will recognise. There is a licence limit on the number of these usernames, so be sure not to use more than one.

It is very important to set a password for your Perforce username, or else anyone can impersonate you. Use the `p4 passwd` command to do this. Once a password is set for your account, you'll need to tell Perforce what it is. You can do this by setting the environment variable `P4PASSWD`, or you can use the `-P` flag with the `p4` command.

There are a few techniques you can use to avoid having to either set an environment variable or type the password on every command. One is to create a shell alias, for example, in `bash`, add something like

```
alias p4='p4 -P secret'
```

to your *.bash\_profile* file. Another way is to create a small shell script, for example

```
#!/bin/bash
```

```
p4 -P secret $@
```

And use this instead of running `p4` directly.

With either of these, be sure the file containing your password (the *.bash\_profile* or shell script file) is only readable by you.

The `p4 users` command lists all currently known users.

Once these three environment variables are set, you can use the perforce `p4` client exactly as described in its documentation. After setting these variables and connecting to the repository for the first time, you should use the `p4 user` and `p4 client` commands to tell perforce the details of your new username and your new client workspace specifications.

## Ending a Repository Session

When you have finished a session using the repository, you should kill off the ssh client process to break the tunnel. Since ssh forked itself into the background, you'll need to use something like ps with the appropriate options to find the ssh process and then kill it manually. The default signal of SIGTERM is fine.

## Overview of the Repository

Please read at least the introductory sections of the Perforce User Guide (and perhaps the Quick Start Guide as well) before reading this section.

Every repository user typically "owns" a "branch" of the mainline code in the repository. They hold the "pumpkin" for things in this area, and are usually the only user who will modify files there. This is not strictly enforced in order to allow the flexibility of other users stealing the pumpkin for short periods with the owner's permission.

Here is the current structure of the repository:

```

/-----+-----perl - Mainline development (bleadperl)
 +-----perlio - PerlIO Pumpkin's Perl
 +-----vmsperl - VMS Pumpkin's Perl
 +-----maint-5.004-----perl - Maintenance branches
 +-----maint-5.005-----perl
 +-----maint-5.6-----perl
 +-----maint-5.6-----pureperl

```

Perforce uses a branching model that simply tracks relationships between files. It does not care about directories at all, so any file can be a branch of any other file—the fully qualified depot path name (of the form `//depot/foo/bar.c`) uniquely determines a file for the purpose of establishing branching relationships. Since a branch usually involves hundreds of files, such relationships are typically specified en masse using a branch map (try `'p4 help branch'`). `'p4 branches'` lists the existing branches that have been set up. `'p4 branch -o branchname'` can be used to view the map for a particular branch, if you want to determine the ancestor for a particular set of files.

The mainline (aka "trunk") code in the Perl repository is under `"//depot/perl/..."`. Most branches typically map its entire contents under a directory that goes by the same name as the branch name. Thus the contents of the perlio branch are to be found in `//depot/perlio`.

Run `'p4 client'` to specify how the repository contents should map to your local disk. Most users will typically have a client map that includes at least their entire branch and the contents of the mainline.

Run `'p4 changes -l -m10'` to check on the activity in the repository. `//depot/perl/Porting/genlog` is useful to get an annotated changelog that shows files and branches. You can use this listing to determine if there are any changes in the mainline that you need to merge into your own branch. A typical merging session looks like this:

```

% cd ~/p4view/perlio
% p4 integrate -b perlio # to bring parent changes into perlio
% p4 resolve -am ./... # auto merge the changes
% p4 resolve ./... # manual merge conflicting changes
% p4 submit ./... # check in

```

If the owner of the mainline wants to bring the changes in perlio back into the mainline, they do:

```

% p4 integrate -r -b perlio
...

```

Generating a patch for change#42 is done as follows:

```

% p4genpatch 42 > change-42.patch

```

*p4genpatch* is to be found in `//depot/perl/Porting/`.



The usual routine to apply a patch is

```
% p4 edit file.c file.h
% patch < patch.txt
```

(any necessary, re-Configure, make regen\_headers, make clean, etc, here)

```
% make all test
```

(preferably make all test in several platforms and under several different Configurations)

```
% while unhappy
do
 $EDITOR
 make all test
done
% p4 submit
```

Other useful Perforce commands

```
% p4 describe -du 12345 # show change 12345
```

Note: the output of "p4 describe" is not in proper diff format, use the *Porting/p4genpatch* to get a diff-compatible format.

```
% p4 diff -se ./... # have I modified something but forgotten
 # to "p4 edit", easy faux pas with autogenerated
 # files like proto.h, or if one forgets to
 # look carefully which files a patch modifies
% p4 sync file.h # if someone else has modified file.h
% p4 opened # which files are opened (p4 edit) by me
% p4 opened -a # which files are opened by anybody
% p4 diff -du file.c # what changes have I done
% p4 revert file.h # never mind my changes
% p4 sync -f argh.c # forcibly synchronize your file
 # from the repository
% p4 diff -sr | p4 -x - revert
 # throw away (opened but) unchanged files
 # (in Perforce it's a little bit too easy
 # to checkin unchanged files)
```

Integrate patch 12345 from the mainline to the maint-5.6 branch: (you have to in the directory that has both the mainline and the maint-5.6/perl as subdirectories)

```
% p4 integrate -d perl/...@12345,12345 maint-5.6/perl/...
```

Integrate patches 12347-12350 from the perl5 branch to the mainline:

```
% p4 integrate -d perl5/...@12347,12350 perl/...
```

## Contact Information

The mail alias <perl-repository-keepers@perl.org> can be used to reach all current users of the repository.

The repository keeper is currently Gurusamy Sarathy <gsar@activestate.com>.

## AUTHORS

Malcolm Beattie, mbeattie@sable.ox.ac.uk, 24 June 1997.

Gurusamy Sarathy, gsar@activestate.com, 8 May 1999.

Slightly updated by Simon Cozens, simon@brecon.co.uk, 3 July 2000.

More updates by Jarkko Hietaniemi, jhi@iki.fi, 28 June 2001.

Perforce clarifications by Randall Gellens, [rcg@users.sourceforge.net](mailto:rcg@users.sourceforge.net), 12 July 2001.