

# BEGINNING PERL

*a language by Larry Wall*

## **Practical Extraction and Report Language or Pathologically Eclectic Rubbish Lister**

*Tom Christiansen*  
*CONVEX Computer Corporation*  
*<tchrist@convex.com>*

Copyright 1990, 1991, 1992, 1993

*There's more than one way to do it!*

TSC - perl.00

Sept 18, 1992

Slide 0

### Goals

- We'll learn to write simple *perl* programs.
- We'll learn to read complex *perl* programs.
- We'll learn “*perl* for C programmers”.
- This is not “everything you ever wanted to know about *perl* but were afraid to ask” — various technical details will be omitted.

TSC - perl.01

Sept 18, 1992

Slide 1

## What is perl?

- Like interpreted C with *sh*, *sed*, and *awk*.
- Perl will be easier if you already know a little bit about those.
- Also derives from at least 15 other UNIX tools.
- Optimized for text, can also handle binary data.
- Especially suitable for system management tasks.
- Its author calls it:
  - “A shell for C programmers.”
  - “The Swiss-army chainsaw of UNIX.”

TSC – perl.02

Sept 18, 1992

Slide 2

## Features

- Easy to get started on due to similarity to familiar UNIX tools.
- Faster program development.
- Faster execution than shell script equivalents.
- More powerful than *sed*, *awk*, or *sh*
- Translators for old *find*, *sed*, and *awk* scripts.
- Comes with a symbolic debugger, written in Perl.
- Portable across many different architectures.
- No arbitrary limits (like string length).
- Fits nicely into UNIX tool-and-filter philosophy.
- It's free!

TSC – perl.03

Sept 18, 1992

Slide 3

## Getting Sources and Info

- Sources from any *comp.sources.misc* archive
- The Frequently Asked Questions list (see reference material) described FTP and UUCP sources.
- From its author, Larry Wall  
<lwall@netlabs.com>
- USENET newsgroup *comp.lang.perl* for discussion, questions, comments, and examples.
- See O'Reilly book *Programming perl* by Larry and Randal Schwartz.

TSC – perl.04

Sept 18, 1992

Slide 4

## Perl Coding

- Executable scripts begin `#!/usr/bin/perl`
- Uses standard `#` for comments.
- No need to use backslash for continuation lines.
- White space (almost) never matters.
- Strings extend to matching quote.
- Statements end in semi-colons.
- Blocks use braces like C.

TSC – perl.05

Sept 18, 1992

Slide 5

## Scalars

- Only data type is a scalar, by context either string, numeric, or boolean.
- Numbers stored internally in double precision.
- Variables not declared, except for locals.
- Default to `undef`, the null string or numeric 0.
- Standard (ANSI) C numeric and string constants recognized.
- Strings may contain arbitrary binary data, including nulls.
- A value of "0", 0, or "" is false; all else is true.

TSC - perl.06

Sept 18, 1992

Slide 6

## Assignment Examples

```
# numeric
$x    = 23;
$y    = 0x77;
$z    = 22.49;
$mol  = 6.02e23;
$myriads = 10_000_000;

# boolean
$too_much = ($count > 10);

# string
$user    = "tchrist";
$message = "\tHey you!\007\n";
```

TSC - perl.07

Sept 18, 1992

Slide 7

## Quoting

- Quoting (almost) like shell: single, double, backtick, and <<HERE documents.

```
$color = 'grey';  
$fox = "the quick, $color fox";  
$fox = "the quick, ${color}ish fox";
```

```
$host = 'hostname';      # note backticks  
chop($host);             # toss newline
```

```
$foo = <<HERE;  
Mom said, "Don't $verb yet."  
HERE
```

TSC - perl.08

Sept 18, 1992

Slide 8

## Special Scalar Variables

- Long list of these. Some are:

```
$0  script name  
$_  default for many operations  
$.  line number of last input  
$$  the current pid  
$?  status of last 'backtick', pipe, or system  
$!  current system error message from errno  
$@  error from last eval, do FILE, or require  
$[  array base, 0 by default; awk uses 1
```

- Many, many more: see reference material.

TSC - perl.09

Sept 18, 1992

Slide 9

## Error Messages

- Good error messages always contain:
  - Name of executing program
  - What you were trying to do (syscall)
  - What you were doing it to (file, pid)
  - Why it didn't work (system error message)

```
print STDERR "$0: can't open $file: $!\n";
```

- The `die` built-in automatically appends the filename and line number (if no `\n` given) and exits non-zero.

```
die "can't open $file: $!";
```

- `warn` is a non-fatal `die`.

TSC – perl.10

Sept 18, 1992

Slide 10

## Aggregates

- Two kinds, determined by bracket type:
  - Numerically-indexed arrays of scalars that work like lists or vectors.  

```
$shopping_list[23] = "peas";
```
  - String-indexed arrays of scalars that work like hash tables, or associative arrays in *awk*.  

```
$phones{"sally"} = "414-248-1234";
```
- Separate namespaces for each type of identifier: scalars, lists, tables, functions, formats, labels, packages, filehandles, dirhandles.
- No (directly-supported) nested aggregates or C structures, but associative arrays often work.

TSC – perl.11

Sept 18, 1992

Slide 11

## Arrays

- These work like lists in *csh*; the empty array (list) is represented by ().
- Subscript with integer values in square brackets;
  - Use \$ for one element: `$list[5]`
  - Use @ for entire array: `@list`
- Arrays are 0-based by default.
- No lists of lists; these are both errors:  
`$a[1,2];` # legal but wrong!  
`$a[1][2];` # not legal syntax

TSC – perl.12

Sept 18, 1992

Slide 12

## Arrays

- `$#list` is the index of highest subscript for the array `@list` and NOT the number of elements.
- In a scalar context, `@list` evaluates to the number of elements in that array.
- Unassigned elements are ok.
- Arrays interpolated into double-quoted strings have spaces between elements by default.  
`print "Shopping list: @items\n";`

TSC – perl.13

Sept 18, 1992

Slide 13

## Array Examples

- Some simple assignment examples:  
`$idx[2] = 3; # set element to 3`  
`@nums = (1, 3, 5); # init whole list`  
`@foo = @bar; # copy @bar`  
`@empty = ( ) ; # clear array`
- Backquote evaluation in an array context returns each line as a separate element, with newline.  
`@lines = `some command`;`  
`chop(@lines);`
- Multiple assignments ok:  
`($foo, $bar, $baz) = ('red', 'blue', 'pink');`  
`($foo, $bar) = ($bar, $foo); # swap`

TSC - perl.14

Sept 18, 1992

Slide 14

## Combining Lists

- When you construct a list out of other lists, you only get one flat list at the end.

```
$n      = 10;
@odd    = ( 1, 3, 5 );
@even   = ( 2, 4, 6 );
@nums   = ( @foo, $n, @bar );

print "@nums\n";
--> 1 3 5 10 2 4 6
print "$nums[0]\n";
--> 1
print "$nums[1]\n";
--> 3
```

TSC - perl.15

Sept 18, 1992

Slide 15



## Combining Unequal Lists

- Extra elements on the RHS are discarded:  

```
($x, $y) = (1..10); # like Pascal '..'  
print "$x $y\n";  
--> 1 2
```
- Extra elements on LHS are set to undef:  

```
($x, $y, $z) = (1);  
print "$x + $y = ", $x + $y, "\n";  
--> 1 + == 1  
  
printf "%d + %d == %d\n",  
       $x, $y, $x + $y;  
--> 1 + 0 == 1
```

TSC - perl.16

Sept 18, 1992

Slide 16

## Array Slices

- Subscripting an array with more than one element is called taking a *slice* of that array.  

```
@foo[0..3] = (5..8);  
@foo[0..3] = @bar[5..8];  
  
@idx1      = (1, 6, 2);  
@idx2      = (8, 1, 4);  
  
@foo[@idx1] = @bar[@idx2];
```
- You may subscript temporary lists, too:  

```
$uid = (getpwnam($somebody))[2];  
print (('a'..'z')[15, 4, 17, 11]);
```

TSC - perl.17

Sept 18, 1992

Slide 17

## Array Operators

- Arrays function as lists; you can add items to or remove them from either end using these operators:
  - **pop**        remove last value from end
  - **push**        adds new values to end
  - **shift**        remove first value from front.
  - **unshift**      adds new values to front
- Examples:

```
push(@list, $bar, 'fred');
push(@list, @rest);
$tos = pop(@list);
while ($arg = shift) { print($arg); }
unshift(@ARGV, 'zeroth arg', 'first arg');
```

TSC – perl.18

Sept 18, 1992

Slide 18

## Associative Arrays

- Much like *awk* arrays.
- Until you start thinking in terms of associative arrays, you're not thinking in Perl. They can often replace lengthy loops or complex algorithms.
- Instead of just integers, these can have any value as an index.
- Subscript by string values with curly braces.

```
$ary{"red"} = 41;
$ary{$food} = 'radish';
```

TSC – perl.19

Sept 18, 1992

Slide 19

## Entire Associative Arrays

- Use `%ary` for the whole array:

```
%foo = ();    # clear out

%save = %current; # copy

# initialize key/value pairs:
%map = ('red', 0x0f, 'blue', 0xf0);

printf "%03x %03x %03x\n",
    $map{'red'}, $map{'green'},
    $map{'blue'},

--> 00f 000 0f0
```

TSC - perl.20

Sept 18, 1992

Slide 20

## Associative Array Accesses

- Use `defined` operator to check for valid value:  
`if ($foo{'bar'})` # might be 0  
`if (defined $foo{'bar'})` # often better
- Remove an entry from associative array with `delete` operator.  
`delete $table{$skilled};`
- Multidimensional arrays may be emulated — the real index will be joined together using `$;` as a separator (by default `^\\`).  
`$loc{$x, $y} = 'troll';`

TSC - perl.21

Sept 18, 1992

Slide 21

## Associative Array Slices

- Use @ for a slice (some but not all) of values from an associative array. This is a list value.

```
printf "%03x %03x %03x\n",
       @map{'red', 'green', 'blue'};
--> 00f 000 0f0

@colors = ('red', 'green', 'blue');
printf "%03x %03x %03x\n",
       @map{@colors};
--> 00f 000 0f0
```

TSC - perl.22

Sept 18, 1992

Slide 22

## Associative Arrays as Sets

- Associative arrays can be used as a set type:

```
# mark each article as already read
for $article (12..15, 21..30, 100..200) {
    $seen{$article}++;
}

# check if we've read a given article
if ( $seen{$first_new} ) { ... }
```

TSC - perl.23

Sept 18, 1992

Slide 23

## Associative Array Operators

- The %array operators **keys**, **values**, and **each** can be used to access pieces of associative arrays.

```
foreach $name ( keys %phones ) {  
    print "$name $phones{$name}\n";  
}  
  
print "Numbers are: ", values %phones;  
  
while (($name, $number) = each %phones) {  
    print "$name $number\n";  
}
```

TSC - perl.24

Sept 18, 1992

Slide 24

## Special Array Variables

- **@ARGV**: command line arguments (watch for \$0)
- **@\_**: default for **split** results and subroutine args
- **%ENV**: is the current environment; assignable for this process and its children.  
`$ENV{'PAGER'} = '/usr/local/bin/less';`
- **%SIG**: is used to set signal handlers.
- Several others; see reference material.

TSC - perl.25

Sept 18, 1992

Slide 25

## Operators

- All of C's operators, except casting and '&' and '\*' used for address operators, '.' and '->' for structure fields. The operators '&', '\*', and '.' have different meanings in perl than they do in C.
- Perl operators have the same associativity and precedence as C, but watch out for non-C operators.
- Exponentiation: \*\*, \*\*= (careful of -2\*\*2 )

TSC - perl.26

Sept 18, 1992

Slide 26

## Operator Precedence

```
(  
++ --  
**  
! ~ unary minus  
+ - .  
<< >>  
file test operators: -r, -w, -x, etc.  
named unary operators: chop, chdir, etc.  
< > <= >= lt gt le ge  
== != <=> eq ne cmp  
&  
| ^  
&&  
||  
..  
?:  
assignment operators: +=, *=, etc.  
'  
list operators: unlink, print, etc.
```

TSC - perl.27

Sept 18, 1992

Slide 27

## String operators

- String concatenation: `., .=`  
`$a = $b . $c . $d;`  
`$x .= "\n";` # opposite of chop
- String repetition: `x, x=`  
`$bar = '-' x 72;` # row of 72 dashes
- String tests: `eq, ne, lt, gt, le, ge`  
`if ($x eq 'foo') {`  
 `if ('boy' lt 'girl') { # always true`
- Use `-w` flag to check for using `==` on strings, since  
`'foo' == 'bar' !`

TSC - perl.28

Sept 18, 1992

Slide 28

## File Test Operators

```
$file = "/tmp/foo";  
  
if (-z $file) { ... } # size 0  
  
if (! -T $file) {  
  die "$file is not a text file!";  
}  
  
$age = -M $file;  
if ($age > 365 && -o $file) { # owner?  
  print "Your file $file is very old\n";  
}
```

TSC - perl.29

Sept 18, 1992

Slide 29

## Magical Operators

- Like C operators with a little extra magic.
- Autoincrement works on strings, with carry:

```
$foo = 'zy';  
print ++$foo, "\n";    # zz  
print ++$foo, "\n";    # aaa
```

- Booleans `&&` and `||` return last value looked at:

```
$pager = $ENV{'PAGER'} || '/usr/ucb/more';
```

TSC - perl.30

Sept 18, 1992

Slide 30

## The Range Operator

- The perl range operator, `..`, has two distinct flavors.
- In an array context, generates a list of enumerated values in the given range:  

```
foreach $i (60..75) { $foo[$i] = $j++; }  
@new = @old[30..50];  
@ltrs = 'aa' .. 'zz';
```
- In a scalar context, emulates `sed` and `awk` line-range operator:  

```
if (1..20) {print;}      # 1,20p  
if (/pat/..eof) {print;} # /pat/, $p
```

TSC - perl.31

Sept 18, 1992

Slide 31



## Control Flow Constructs

```
EXPR;  
EXPR MODIFIER EXPR;  
LABEL BLOCK  
  
do BLOCK;           # note semi-colon  
do BLOCK MODIFIER; # not real loops!  
  
if (EXPR) BLOCK  
elsif (EXPR) BLOCK # many of these  
else BLOCK  
  
LABEL while (EXPR) BLOCK  
  
LABEL for (EXPR; EXPR; EXPR) BLOCK  
LABEL foreach $VAR (LIST) BLOCK
```

TSC - perl.32

Sept 18, 1992

Slide 32

## Control Flow

- A statement always has a semi-colon at the end.
- Unlike C, blocks always require curly braces.
- **unless** and **until** are just **if** and **while** negated.
- A MODIFIER is one of **if**, **unless**, **while**, and **until**.

```
die "bad x: $x" unless $x > 0;  
print "$i\n"    until $i++ > $j;
```

TSC - perl.33

Sept 18, 1992

Slide 33

## Logicals for Control Flow

- **||** and **&&** are often used for control flow as in the shell.

```
open (FILE,$file) || die "$file: $!";  
unlink('bar')      && print "ok\n";
```

- They evaluate left to right, using short-circuit logic, and return the last expression evaluated.

TSC – perl.34

Sept 18, 1992

Slide 34

## Loop Control

- Use **last** and **next** rather than C's **break** and **continue**.
- **redo** restarts current iteration, ignoring loop test.
- Labels are optional identifiers to label blocks for use with **next**, **last**, and **redo**.
- Changing the index variable in a **foreach** loop changes the array.
- **for** and **foreach** are completely synonymous.

TSC – perl.35

Sept 18, 1992

Slide 35

## For and Foreach

- The C way, with fancy loop control:

```
LOOP1: for ($i = 0; $i < @ary1; $i++) {  
    for ($j = 0; $j < @ary2; $j++) {  
        if ($ary1[$i] > $ary2[$j]) {  
            next LOOP1;  
        }  
        $ary1[$i] += $ary2[$j];  
    }  
}
```

- The perl way, without subscripts:

```
LOOP2: foreach $i (@ary1) { # more idiomatic  
    for $j (@ary2) {  
        next LOOP2 if $i > $j;  
        $i += $j;  
    }  
}
```

TSC - perl.36

Sept 18, 1992

Slide 36

## Case Statements

- In Perl, you roll your own **switch** statement.

```
$_ = ".....";  
SWITCH: {  
    /foo/ && do {  
        &something();  
        last SWITCH;  
    };  
    /bar/ && do {  
        &something_else();    # FALLTHRU!  
    };  
    /glarch/ && do {  
        if ($xyzyzy) {  
            $_ = &something_funky();  
            redo SWITCH;  
        }  
    };  
    # default code here  
}
```

TSC - perl.37

Sept 18, 1992

Slide 37

## Regular Expressions

- Perl's regular expressions are a superset of nearly all other commonly used regexps.
- ANSI C escapes: `\t`, `\n`, `\a`, `\033`, `\x1b`
- Take *egrep* and add these (capitals negate):
  - `\w`, `\W`: alphanumeric or underbar
  - `\d`, `\D`: one digit
  - `\s`, `\S`: space [ `\t\n\r` ]
  - `\b`, `\B`: boundary
- Don't escape these for special meaning:  
( ) | + ? { }
- Pattern matching variables (expensive):
  - `$'` text before the match
  - `$&` the entire match itself
  - `$'` text after the match

TSC - perl.38

Sept 18, 1992

Slide 38

## Regexp Switches

- Use `=~` and `!~` if variable to search isn't `$_`.  

```
if ($msg =~ /^Error/) { ... }  
if ($msg !~ /^Error/) { ... }
```
- `/i` switch on pattern matching means be case-insensitive.  

```
if (/red/i) { # red, RED, Red }
```
- Variables can interpolate into regexps.  

```
if ( /$my_name/ ) { ... }
```

TSC - perl.39

Sept 18, 1992

Slide 39

## Regular Expression Example

```
$foo = 'Often I saw Ten Bears on the hill.';

$foo =~ /\bten\b/i;

print "Before:  =${'=\n'};
print "Match:   =${&=\n'};
print "After:   =${'=\n'};

-->

Before:  =Often I saw =
Match:   =Ten=
After:   = Bears on the hill.
```

TSC - perl.40

Sept 18, 1992

Slide 40

## Subexpression Matching

- Match selected subexpressions for easy parsing. Use `\1 .. \9` inside regexps, `$1 .. $9` outside of them.

```
print "$1 repeats\n" if /(\w+)\1/;

if (/(\d\d):(\d\d):(\d\d)/) {
    $hours    = $1;
    $minutes  = $2;
    $seconds  = $3;
}
```

- In array context, returns the matches as array:

```
($hours, $minutes, $seconds) =
    /(\d\d):(\d\d):(\d\d)/;
```

TSC - perl.41

Sept 18, 1992

Slide 41

## Substitution

- Use `s/old/new/` for substitutions; the `=~` operator required except for `$_`.  
`s/red/blue/i;`  
`s/^(\\S+)\\s+(\\S+)/$2 $1/; # swap words`
- Returns the number of successful substitutes:  
`$changes = ($address =~ s/$old/$new/g);`
- Often applied to an assignment statement:  
`($basename = $0) =~ s#.#/##;`
- `/e` means evaluate the RHS as an expression:  
`$_ = "7abc123xyz\\n";`  
`s/\\d+/4 * $%/eg; # 28abc492xyz`

TSC - perl.42

Sept 18, 1992

Slide 42

## Translation

- Perl can perform translations like the `tr` program or `sed`'s `y` operator (`tr` same as `y`).
- Unlike matching and substitution, variables don't interpolate.
- Returns the number of characters translated.  
`tr/a-z/A-Z/; # toupper`  
`y/\\000-\\037\\177-377/?/;`  
`$line =~ tr/A-Za-z/N-ZA-Mn-za-m/; # rot13`  
`($lower = $upper) =~ tr/A-Z/a-z/;`  
`$slash_count = ($path =~ y#/#/#);`

TSC - perl.43

Sept 18, 1992

Slide 43

### Regexp Examples

- Assume each line has a user name and a count of pages used.

- Include sample input with any parsing code.

```
# Input format:
#      TCHRIST 20
#      ROOT    7
#      DAEMON  2
if (/(\w+)\s+(\d+)/) {
    ($user, $count) = ($1, $2);
    $user =~ tr/A-Z/a-z/;
    $pages_used{$user} += $count;
    s/^/current total:/;
    print;
}
```

TSC - perl.44

Sept 18, 1992

Slide 44

### I/O

- Perl I/O is often line-oriented.
- Filehandles have their own namespace; typically are in upper case to protect against future keywords.
- Newlines are always left intact, but may be chopped as needed.
- Mentioning a filehandle in angle brackets reads next line in a scalar context, all lines in an array context:

```
$line = <TEMP>; # read line
chop($line = <TEMP>); # read and chop
@lines = <TEMP>; # read all lines
chop(@lines); # chop all lines
```

TSC - perl.45

Sept 18, 1992

Slide 45

### Predefined Filehandles

- **STDIN, STDOUT, and STDERR** are always pre-opened.
- **DATA** is everything past **\_\_END\_\_** in script.
- Reading from null filehandle **<>** successively reads from all files supplied on **@ARGV** (or **STDIN** if empty) as one virtual filehandle. When used this way, **\$ARGV** is the current filename.

TSC - perl.46

Sept 18, 1992

Slide 46

### I/O Examples

- When used by itself in a **while** construct, input lines are automatically assigned to the **\$\_** variable.  

```
while (<INPUT>) { # assign to $_
    chop;         # discard newline
    ....
}
```
- The **\$/** variable is the record separator. It must be a string, not a regular expression. Set to **' '** for paragraph reads, or to **undef** to read entire file into a scalar variable.

TSC - perl.47

Sept 18, 1992

Slide 47



## I/O Examples (cont)

- Here are 4 ways of writing *cat* in *perl*:

```
while ($_ = <>) { print $_; }  
while (<>) { print; }  
print while <>;  
print <>; # not quite equiv
```

- Filter example. Note <> use.

```
while ( <> ) {           # read into $_  
    next if /^#/;        # skip comments  
    s/left/right/g;      # global substitute  
    tr/A-Z/a-z/;         # tolower  
    print "$ARGV:";      # filename  
    print;               # print the line  
}
```

TSC - perl.48

Sept 18, 1992

Slide 48

## Opening Files

- If not using the pseudo-file <>, open a filehandle:

```
open(PWD, "/etc/passwd");      # read  
open(TMP, ">/tmp/foobar.$$");   # write  
open(LOG, ">>logfile");         # append  
open(TOPIPE, "| lpr");         # pipe out  
open(FROMPIPE, "netstat -a |"); # pipe in
```

- Open returns true on success (child pid for pipes), false on failure.
- Don't forget to close pipes to flush them, and beware forking with full buffers.

TSC - perl.49

Sept 18, 1992

Slide 49

## Database Access

- Associative arrays may be bound to dbm files with `dbmopen()`.  

```
dbmopen(%alias, '/usr/lib/aliases', 0444)
|| die "$0: can't dbmopen aliases: $!";
```
- May have to add a null byte to keys or chop it from the values for some text dbm files created by C programs to keep them happy.
- Use `each` not `keys` for big dbm files.  

```
$alias = $alias{"postmaster\0"};
chop $alias;
print "postmaster -> $alias;\n";
```

TSC - perl.50

Sept 18, 1992

Slide 50

## DBM Example

- `countman`: determine number of entries in man tree.  

```
for (split(/:/, $ENV{MANPATH} || '/usr/man')) {
    if (!dbmopen(%whatism, "$_/_whatism", undef)) {
        warn "$0: dbmopen $_: $!"
        next;
    }
    print $_, ': ';
    while (each %whatism) { $count++; }
    print $count, "\n";
}
```

TSC - perl.51

Sept 18, 1992

Slide 51

## Fancy I/O

- All system I/O functions available, including:
  - Access to `getc`, `eof`, `seek`, `close`, `flock`, `ioctl`, `fcntl`, `pipe`, and `select` calls. These take filehandles.
  - Access to `mkdir`, `rmdir`, `chmod`, `chown`, `link`, `symlink`(if supported), `stat`, `readlink`, `rename`, and `unlink` calls for use with filenames.
- Pass `print` or `printf` a filehandle (no comma) unless printing to `STDOUT`:

```
printf LOG "%-8s %s: weird bits: %08x\n",  
          $program, &ctime, $bits;
```

TSC - perl.52

Sept 18, 1992

Slide 52

## System Functions

- A plethora of functions from the C library provided as built-ins, including most system calls. These include: `alarm`, `chdir`, `chroot`, `exec`, `exit`, `fork`, `getlogin`, `getpgrp`, `getppid`, `kill`, `setpgrp`, `setpriority`, `sleep`, `syscall`, `system`, `times`, `umask`, `utime`, `wait`, and `waitpid`.
- If your system has Berkeley-style networking: `bind`, `connect`, `send`, `getsockname`, `getsockopt`, `getpeername`, `recv`, `listen`, `socket`, `socketpair`.
- If C library supports them: `getpw*`, `getgr*`, `gethost*`, `getnet*`, `getserv*`, and `getproto*`.

TSC - perl.53

Sept 18, 1992

Slide 53

## System

- To run an external command, use the **system** function.
- **/bin/sh** will be called if metacharacters are in the string.
- Return value is exit value of the command <<8. This is also stored in the **\$?** variable.
- Use a list to avoid the shell entirely.

```
system "ls -F /tmp > $tmpfile 2>/dev/null";  
system "touch $file";  
system $editor, $file; # no shell
```

TSC - perl.54

Sept 18, 1992

Slide 54

## Directory Access, Shell style

- Easiest way to write is using built-in file globbing notation.
- A string enclosed in angle brackets with shell metacharacters is handed off to the shell for expansion.

```
chmod 0444, <*.c>;  
chdir '/tmp';  
foreach $file ( <*. [ch]> ) {  
    print $file, "\t", -s $file, "\n";  
}
```

- Because the shell is (currently) used, you don't get dot files unless you ask for them.  
**@files = <\*. [ch] .#\*. [ch]>;**

TSC - perl.55

Sept 18, 1992

Slide 55

## Directory Access, C style

- Sometimes your `<*>` globs can blow up.
- The directory-reading routines are provided as built-ins and operate on directory handles in their own namespace.
- Supported routines are `opendir`, `readdir`, `closedir`, `seekdir`, `telldir`, and `rewinddir`.
- `readdir` returns one value in a scalar context, all in an array one, analogous to `<FILE>`.

TSC - perl.56

Sept 18, 1992

Slide 56

## Directory Access Example

- `straycats`: finds catpages without parent manpage

```
for $root (split(/:/, $ENV{MANPATH} || '/usr/man')) {  
  chdir($root) || die "can't chdir to $root: $!\n";  
  foreach $catdir ( <cat*> ) {  
    unless (opendir (CATDIR, $catdir) {  
      warn "can't opendir $catdir: $!";  
      next;  
    }  
    ($mandir = $catdir) =~ s/cat/man/;  
    foreach $file ( readdir(CATDIR) ) {  
      next if $file eq '.';  
      next if $file eq '..';  
      next if -e "$mandir/$file";  
      print "no man page for $root/$catdir/$file\n";  
    }  
  }  
}
```

TSC - perl.57

Sept 18, 1992

Slide 57

## User-Defined Subroutines

- Subroutines called as `&subr($arg1, $arg2)`.
- Order of definition doesn't matter.
- Return value is from `return` statement or last expression in block.
- Functions may be called recursively or indirectly.

```
$foo = 'bar';  
&$foo();           # calls &bar()
```

TSC - perl.58

Sept 18, 1992

Slide 58

## Parameter Passing

- Scalars, lists, and arrays may all be passed as parameters. Parameters are received by the subroutine in the special array `@_`.
- Parameters are passed by reference, so it's wise to copy into local variables before changing them.
- Multiple lists or arrays collapse into one list.

```
&clist(@a, @b, @c);  
sub clist {  
    # careful: LHS is list  
    local($count) = scalar(@_);  
    print "received $count elements\n";  
}
```

TSC - perl.59

Sept 18, 1992

Slide 59

## Subroutine Examples

```
sub square { $_[0] * $_[0]; }
$x = &square(1.43);

sub double_all {
    local(@list) = @_;
    for (@list) { $_ *= 2; }
    @list;
}

sub double_inplace {
    for (@_) { $_ *= 2; }
}

@list = (1..10);
@twice = &double_all(@list);
&double_inplace(@twice);
```

TSC - perl.60

Sept 18, 1992

Slide 60

## Local Variables

- Use local variables to avoid stomping on your callers' variables.

```
$name = '/tmp/foo.c';

sub basename {
    local($name) = $_[0];
    $name =~ s!.*!/!!;
    $name;
}

print &basename($name);
--> foo.c

print $name;
--> /tmp/foo.c
```

TSC - perl.61

Sept 18, 1992

Slide 61

## Local Scoping Caveat

- Scope of a `local` variable is through end of its enclosing block.
- Storage released when block is finally exited. This makes a big difference in efficiency:

```
for ($i = 1; $i < 1e6; $i++) {  
    # million @foos before we're done!  
    local(@foo);  
}  
  
local(@foo);  
for ($i = 1; $i < 1e6; $i++) {  
    undef @foo;  
}
```

TSC - perl.62

Sept 18, 1992

Slide 62

## Dynamic Scoping

- Perl is *dynamically scoped*, which means that a `local()` in no way stops your callees from seeing (and touching) your variables if they want.

```
$name = '/tmp/foo.c';  
sub basename {  
    local($name) = $_[0];  
    $name =~ s!.*!/!!;  
    &bang();  
    return $name;  
}  
sub bang { $name =~ s/$!/!/; }  
print &basename($name);  
--> foo.c!  
print $name;  
--> /tmp/foo.c
```

TSC - perl.63

Sept 18, 1992

Slide 63



## Passing by Name

- A special notation exists for referring to all types of some name: `*foo` means all objects of the name `foo`.
- This is more efficient than passing long lists.
- Preserves identity of separate arrays.
- Only way to declare local filehandles.
- Occludes *all* types of that name, so be careful.

TSC - perl.64

Sept 18, 1992

Slide 64

## Pass by Name Example

- Operate on two arrays separately:

```
sub add2 {
    local(*a,*b) = @_;
    local($i, $sum, $max);

    $max = $#a; $max = $#b if $#b > $#a;
    for ($i = 0; $i < $max; $i++)
        $sum += ($a[$i]* + $b[$i]*$b[$i])/2;
    }
    $sum;
}
@x = (1..10);
@y = (5..50);
print &add2(*x, *y);
--> 20390
```

TSC - perl.65

Sept 18, 1992

Slide 65

## Signal Handling

- The %SIG array is used to set a signal handler by name.
- The function will get the name of the signal as its argument.
- I use upper-case for these because they're called asynchronously.
- You may also choose the actions 'DEFAULT' and 'IGNORE'.

TSC - perl.66

Sept 18, 1992

Slide 66

## Signal Example

- For example:

```
$SIG{'INT'} = 'CLEANUP';  
$SIG{'HUP'} = 'CLEANUP';  
$SIG{'QUIT'} = 'CLEANUP';  
$SIG{'TERM'} = 'CLEANUP';  
  
sub CLEANUP {  
    warn "$0: caught SIG$_[0]!\n";  
    unlink $tmp;  
    exit -1;  
}
```

TSC - perl.67

Sept 18, 1992

Slide 67

## Scalar Built-in Functions

- Numeric functions: `cos`, `sin`, `atan2`, `log`, `exp`, `rand`, `srand`, `oct`, `hex`, `int`.
- Time functions: `time`, `times`, `localtime`, `gmtime`.
- String functions: `chop`, `crypt`, `index`, `rindex`, `length`, `substr`, `sprintf`.
- Many of these default to an argument of `$_`.

TSC - perl.68

Sept 18, 1992

Slide 68

## Substr

- `substr` takes a scalar value, a start position, and an optional length. A negative position starts from the rear.

```
$sentence = 'In Xanadu did Kubla Khan';  
$first    = substr($sentence, 0, 5);  
--> In Xa  
  
$middle    = substr($sentence, 10, 10);  
--> did Kubla  
  
$rest      = substr($sentence, 16);  
--> bla Khan  
  
$end       = substr($sentence, -5);  
--> Khan
```

TSC - perl.69

Sept 18, 1992

Slide 69

### Assignments to substr

- **substr** acts as an lvalue, so can be assigned to:

```
$sentence = 'In Xanadu did Kubla Khan';  
  
substr($sentence, 3, 0) = 'distant '  
--> In distant Xanadu did Kubla Khan  
  
# notice use of '=' instead of '~'  
substr($sentence, -5) =~ tr/a-z/A-Z/  
--> In distant Xanadu did Kubla KHAN
```

TSC - perl.70

Sept 18, 1992

Slide 70

### Array Built-Ins: **split**

- **split** breaks up a scalar into a list based on a regexp. Normal usage is:  
`@list = split(/PATTERN/, $string);`
- Returns number of matches in a scalar context.
- By default, uses `$_` for input, `@_` for output, and `/\s+/` as the delim.
- See reference material for other fancy **split** features.

TSC - perl.71

Sept 18, 1992

Slide 71

## Split Example

- Count all distinct words:

```
while (<>) {  
    for (split) {  
        next unless /^\\w+$/;  
        $count{$_}++;  
    }  
}  
  
for (keys %count) {  
    printf "%5d %s\\n", $count{$_}, $_;  
}
```

TSC – perl.72

Sept 18, 1992

Slide 72

## Array Built-Ins: split and join

- Here's one way to get the password:

```
while (<PASSWD>) {  
    ($login, $passwd, $uid, $gid, $gcos,  
     $home, $shell) = split(/:/);  
}
```

- Here's another:

```
$passwd = (split(/:/, $_))[1];
```

- The inverse of split is join.

```
$line = join(':', $login, $passwd, $uid,  
             $gid, $gcos, $home, $shell);
```

TSC – perl.73

Sept 18, 1992

Slide 73

### Array Built-Ins: **grep**

- **grep** returns a new list consisting of elements for which the expression was true (or the count of them in a scalar context):

```
@lines = grep(!/^#/ , @lines);  
$count = grep(!/^#/ , @lines);
```

- Since **grep** has a reference to each array elements, it is sometimes used as an implicit loop:

```
grep(s/^#/, @lines);
```

TSC – perl.74

Sept 18, 1992

Slide 74

### Array Built-Ins: **reverse**

- **reverse** inverts a list:

```
foreach $tick (reverse 0..10) {  
    print "$tick...";  
}  
print "bang!\n";
```

- In a scalar context, reverses the string:

```
$is_palindrome = reverse($foo) eq $foo;
```

TSC – perl.75

Sept 18, 1992

Slide 75

### Array Built-Ins: **sort**

- **sort** returns a new list with elements ordered according to their ASCII values.
- May use a named array or any arbitrary list.  

```
@a = ("rat", "dog", "cat", "mouse");  
@a = sort @a;  
--> cat dog mouse rat  
@a = reverse sort @a  
--> rat mouse dog cat  
print join(", ", sort @b, @c, @d), "\n";
```

TSC – perl.76

Sept 18, 1992

Slide 76

### Customized Sorting

- For customized sorting, declare your own subroutine and pass its name (or a variable with its name) as the first parameter to **sort**, or inline the sort code anonymously.
- This routine will be called repeatedly for each pair of values in the list, and should return -1, 1, or 0.
- The values to be compared are passed in as **\$a** and **\$b**.
- The special sort operators **<=>** and **cmp** can make this easier.

TSC – perl.77

Sept 18, 1992

Slide 77

## Customized Sorting Examples

```
@list = (43, 8, 19, 2);  
@newlist = sort @list;  
print "(", join(", ", @newlist), ")\n";  
--> (19, 2, 43, 8)  
  
sub numerically { $a <=> $b; }  
@newlist = sort numerically @list;  
print "(", join(", ", @newlist), ")\n";  
--> (2, 8, 19, 43)  
  
sub descending { $b <=> $a; }  
@newlist = sort descending @list;  
print "(", join(", ", @newlist), ")\n";  
--> (43, 19, 8, 2)
```

TSC - perl.78

Sept 18, 1992

Slide 78

## Inline Sorting Examples

```
@list = (43, 8, 19, 2);  
@newlist = sort @list;  
print "(", join(", ", @newlist), ")\n";  
--> (19, 2, 43, 8)  
  
@newlist = sort { $a <=> $b; } @list;  
print "(", join(", ", @newlist), ")\n";  
--> (2, 8, 19, 43)  
  
@newlist = sort { $b <=> $a; } @list;  
print "(", join(", ", @newlist), ")\n";  
--> (43, 19, 8, 2)
```

TSC - perl.79

Sept 18, 1992

Slide 79



## Sorting Associative Arrays

- Because %arrays are in random order, you often sort them:

```
foreach $key (sort keys %ENV) {  
    print $key, '=', $ENV{$key}, " n";  
}
```

- Sometimes you want to sort on the values, not the keys:

```
for (sort {$map{$a} <=> $map{$b}} keys %map) {  
    print "$_\t$map{$_}\n";  
}
```

TSC - perl.80

Sept 18, 1992

Slide 80

## One Liners

- You can write great one-liners in *perl* using a few command-line switches:

- **-p** print each line
- **-n** DON'T print each line
- **-e** Supply an expression
- **-i.bak** In-place edit

```
# useful at end of 'find foo -print'  
perl -ne 'chop;unlink'
```

```
# change all foo to bar in-place  
perl -pei.bak 's/\bfoo\b/bar/g' *.c
```

```
# reverse input line  
perl -e 'print reverse <>'
```

TSC - perl.81

Sept 18, 1992

Slide 81

### A More Detailed Example

- This program checks `.rhosts` files for lines containing only a plus or a minus.

```
1  open (PASSWD, '/etc/passwd')
2  || die "$0: can't open passwd: $!";
3  while (<PASSWD>) {
4      ($login, $passwd, $uid, $gid,
5          $gcos, $home, $shell) = split(/:/);
6      open (RHOSTS, "$home/.rhosts") || next;
7      $found = 0;
8      while (<RHOSTS>) {
9          $found |= /^[+-]$/;
10         last if $found;
11     }
12     next unless $found;
13     system "Mail -s \"You have a bad \" .
14         ".rhosts\" $login </dev/null";
15 }
```

TSC - perl.82

Sept 18, 1992

Slide 82

### A Few Lingering Problems

- Assumes `passwd` file has all the entries. Better to use `getpwent` instead.
- It would be nice to know what host the problem was on.
- That's not a very informative mail message.
- Do we care about shells that aren't in `/etc/shells`?
- The program should output the problems it encountered so a summary report can be generated.

TSC - perl.83

Sept 18, 1992

Slide 83

## Tidying Up

- Since we really shouldn't be parsing the passwd file by hand, here's a better way to do it:

```
while ( ($name,$passwd,$uid,$gid,$quota,  
        $comment,$gcos,$dir,$shell) =  
        getpwent)  
  
    { ... }
```
- Here's how to get the hostname:

```
$host = `hostname`;  
chop $host;
```
- Or more concisely:

```
chop($host = `hostname`);
```

TSC - perl.84

Sept 18, 1992

Slide 84

## Sending Nicer Mail

```
($name = $gcos) =~ s/[\s,].*//;  
  
print "$login@$host\n"; # for log  
open (MAILER, "| Mail -s \"security hazard\" $login");  
print MAILER <<EO_MESSAGE;  
Dear $name,  
  
On the machine $host, you have a $_ in "$rhosts".  
This is a security hazard. Please replace this with  
an explicit list of trusted hosts.  
  
Thank you.  
  
[This mail generated by the $program program]  
EO_MESSAGE  
close MAILER;
```

TSC - perl.85

Sept 18, 1992

Slide 85

## Checking for a Good Shell

```
$SHELLS = "/etc/shells";  
open SHELLS || # use $SHELL  
    die "$0: couldn't open $SHELLS: $!";  
while (<SHELLS>) {  
    next if /^#/;  
    chop;  
    ++$shells{$_};  
}  
close SHELLS;
```

- Add this line after each passwd entry fetch;  
    **next unless \$shells{\$shell};**
- See appendices for a complete version of the *bdrhosts* program.

TSC - perl.86

Sept 18, 1992

Slide 86

## Perl Gotchas

- Forgetting your dollar signs: **\$foo = bar;**
- Saying **@foo[1]** instead of **\$foo[1]**
- Different behavior in scalar context in array context.
- Not using **-w** to find typos.
- Dynamic scope.
- Not chopping the newline off of **`commands`**.
- Making operators behave like functions, and forgetting: **print (3+2)\*4;**

TSC - perl.87

Sept 18, 1992

Slide 87

### Topics Deserving More Time

- ● There are well over a dozen little topics that each warrant their own slide, or more:
  - Communicating via pipes
  - Bit manipulation using **vec**
  - Interfacing with C programs
  - Setuid scripts
  - The *perl* library
  - The debugger
  - Formatted I/O
  - Using and for **ioctl**, **fnctl**, and **syscall**.

TSC - perl.88

Sept 18, 1992

Slide 88

### More Topics Deserving More Time

- **eval** for setjmp/longjmp, code generation, and exception handling
- **pack** and **unpack**
- Networking
- Both forms of the **select** call.
- More complex data structures
- Pointers in perl
- Custom system calls
- Packages

TSC - perl.89

Sept 18, 1992

Slide 89